Implementing LOTOS as Asynchronously Communicating Processes

by

Peter Sjödin
Implementing LOTOS as Asynchronously Communicating Processes

Peter Sjödin
Swedish Institute of Computer Science
Box 1263, S-16428 Kista, Sweden
peter@sics.se

SICS Research Report SICS/R-90/9014, November 1990

Abstract

A technique is presented for translating LOTOS specifications into implementations executing as asynchronously communicating processes. This generation of implementations is described as transformations of LOTOS expressions. A protocol for implementing LOTOS synchronisation is described.

1 Introduction

We present an approach for deriving implementations from formal specifications. We focus on the formal description technique LOTOS [IS8], a specification language primarily intended for specifying communication protocols in open systems. The reader is assumed to have some familiarity with LOTOS, and we refer to the ISO standard document [IS8] for an introduction to the language.

An important property of a formal specification of a system such as a communication protocol, is that it is a functional description of the system. A functional description should be implementation independent so that the specification is applicable to all potential implementations of the system. In other words, a specification should describe what a system does, but not how the system should do it. The generation of an implementation from a specification can be regarded as a process of stepwise refinement, where a functional specification is transformed into an implementation specification. An implementation specification contains enough detailed information to make it possible to directly realise the system (by for example translating it to machine code).

Ideally, the transformation of a functional specification into an implementation specification should be done automatically, by a tool. This is unrealistic, since many design decisions requiring human interaction have to be made during the transformation. For example, it has to be decided what algorithms should be used, how resources should be allocated (such as partitioning the system into processes in a distributed environment), and how services provided by the implementation environment (such as operating service primitives) should be used. A more realistic scenario is a semi-automatic transformation, where the specification is gradually modified by a designer (with help of some tools). After
a modification the designer can check that the modified system is consistent with the original specification (this checking can also be done by some tool). When the specification has been refined enough to qualify as an implementation specification, it can be compiled into an implementation. We regard an implementation to be a program executing in an implementation environment, where the implementation environment provides some services to the program, for example access to a file system and primitives for process communication.

The compilation of an implementation specification into an implementation is non-trivial if the implementation environment does not provide support for concepts of the specification language. The main contribution of this paper is to define a transformation from implementation specifications in a subset of LOTOS to implementations in an environment only supporting asynchronous communication between processes. We define a subset of LOTOS called intermediate LOTOS for representing implementation specifications. We specify in LOTOS the services provided by an asynchronous implementation environment, and define another LOTOS subset, run-time LOTOS, which is used to specify programs executing in such environments. We then show how intermediate LOTOS can be implemented by defining a transformation from intermediate LOTOS to run-time LOTOS. This transformation is of such nature that it can easily be automated. We will not consider the problem of transforming a functional specification into an implementation specification.

In an asynchronous implementation environment, process communication is buffered: one process sends a message by storing it in an output buffer. The message will then, at some later time, be moved from the output buffer to the input buffer of the receiving process, which eventually will fetch the message from its input buffer. This process communication scheme is fundamentally different from the process communication scheme in LOTOS, which is is synchronous and atomic. In LOTOS, a communication event can involve any number of processes, and all participants will participate in the interaction at the same time. Further, communication events take place at gates. Several communication events can be possible at a time, but only one of the events will occur. The choice of which event will occur is made non-deterministically. This is illustrated with the following example:

\[
\begin{align*}
\text{process A}[g]: & \text{ exit := } \\
& \quad g \ ?x: \text{ integer; exit} \\
& \quad [] \\
& \quad g \ ?b: \text{ boolean; exit} \\
\text{endproc}
\end{align*}
\]

\[
\begin{align*}
\text{process B}[g]: & \quad \text{ exit := } g \ !10; \text{ exit endproc}
\end{align*}
\]

\[
\begin{align*}
\text{process C}[g]: & \quad \text{ exit := } g \ !\text{false; exit endproc}
\end{align*}
\]

Process A is prepared to input either an integer value or a boolean value at gate \( g \) and then terminate ([] is the LOTOS choice operator, and \( \text{exit} \) denotes the termination of a process). Process B is prepared to output an integer at the same gate, and process C is prepared to output a boolean. Two communication events are thus possible: between A and B, and between A and C. Only one of them will occur, and the processes have no control of (or can make any assumptions about) which event actually will occur in an
execution.

In summary, our approach for implementing LOTOS in an asynchronous environment is to implement LOTOS gates as particular processes, called ports. In order to establish a communication event, processes communicate with ports according to a protocol. We will only consider internal communication within a system, "hidden events", and do not consider communication with the external world.

The protocol we suggest is related to distributed algorithms for synchronising CSP [Hoa85] processes. The main difference between CSP and LOTOS is that CSP communication involves exactly two processes and that it is direct, that is, a CSP process directly addresses by name the processes with which it is prepared to communicate. One of the first general algorithms for CSP (general in the sense that it does not impose any restrictions on the source program) is due to Bernstein [Ber80]. This algorithm has the drawback of a potential "livelock" (it is not guaranteed to terminate), which was eliminated by Buckley and Silberschatz [BS83]. In their algorithm the negotiation is performed in two phases, and Back et. al. [BEKS84] improved the algorithm by making it a single phase algorithm. Ramesh [Ram87b] suggested a similar, but more efficient, algorithm. In [Ram87a], this algorithm is generalised to cover synchronisation of an arbitrary number of processes.

In Section 2, intermediate and run-time LOTOS is described. Section 3 gives the transformation from intermediate LOTOS to run-time LOTOS. In Section 4, a protocol for implementing LOTOS synchronisations is presented. Section 5 discusses how data values can be incorporated in the execution model. Section 6 presents related work and Section 7, finally, summarises the paper and outlines future directions.

2 Intermediate LOTOS and run-time LOTOS

The derivation of an implementation from a LOTOS formal specification consists of several transformation steps, as shown in Figure 1. A LOTOS specification is refined into an implementation specification in intermediate LOTOS. The implementation specification is transformed into run-time LOTOS. Each construct in run-time LOTOS has a direct correspondence in the implementation environment. For example, parallel composition in run-time LOTOS corresponds to dynamic creation of tasks in the implementation environment. A run-time LOTOS specification can be compiled directly into the target programming language.

![Diagram](diagram.png)

Figure 1: Implementation of a system from a LOTOS specification
2.1 Formal semantics of LOTOS behaviour expressions

A LOTOS specification defines a transition system, which is a tuple \( < S, s_0, A, T > \). \( S \) is a set of system states, \( s_0 \) is an initial state (\( s_0 \) is a member of \( S \)), and \( A \) is a set of actions. \( T \) is a set of transitions between states. Each transition is labeled by an action, so a transition is a member of \( S \times A \times S \). For example, assume that \( s_i \) and \( s_j \) are states, \( a \) an action and \( s_i \xrightarrow{a} s_j \) a transition. This means that the system can when in state \( s_i \), do a transition to the state \( s_j \). In the transition the action \( a \) takes place.

In LOTOS a state is described by a behaviour expression. For example, \( a;B_1 \parallel b;B_2 \) is a behaviour expression that describes a state with two possible transitions. Either \( a \) takes place with a transition to the state described by behaviour expression \( B_1 \), or \( b \) takes place with a transition to the state described by behaviour expression \( B_2 \).

The correspondence between transition systems and LOTOS behaviour expressions is defined in [IS8] by a derivation system, consisting of axioms and inference rules. For example, the semantics of the \( \parallel \) operator (choice) is defined by the following two inference rules:

\[
\begin{align*}
B_1 \xrightarrow{a} B'_1 & \quad \Rightarrow \quad B_1 \parallel B_2 \xrightarrow{a} B'_1 \\
B_2 \xrightarrow{a} B'_2 & \quad \Rightarrow \quad B_1 \parallel B_2 \xrightarrow{a} B'_2
\end{align*}
\]

The interpretation of this is that if there is a possible transition from the state described by behaviour expression \( B_1 \) to the state described by \( B'_1 \), then there is also a possible transition from the state described \( B_1 \parallel B_2 \) to the state described by \( B'_1 \), labeled by the same action (and correspondingly for \( B_2 \)).

The semantical definitions of action prefix and process termination of LOTOS form the axioms of the derivation system. Action prefix is for example defined by the following axiom, which says that there is a transition from the state described by \( a;B \) to the state described by \( B \), labeled by \( a \):

\[
a;B \xrightarrow{a} B
\]

2.2 Intermediate representation

Intermediate LOTOS contains all expressions of full LOTOS, with the following exceptions: sequential composition (enable, accept, and exit), so process definitions in intermediate LOTOS will have functionality noexit, process disruption (disable), and generalised parallel and choice expressions over gates. These restrictions are introduced to simplify the mapping to run-time LOTOS and, in many cases, these expressions can be transformed into intermediate LOTOS expressions with equivalent behaviour.

Furthermore, internal events "i" and generalised choice over values (and thus value generating interactions) are not allowed. We regard these constructs to be abstractions, meaning that some aspects of the system are unspecified (not explicitly described): Internal events represent a change of state due to some unspecified behaviour, and choice over values represent binding of some unspecified values to identifiers. In our opinion, specifications containing such constructs are not detailed enough to be directly implementable.
To simplify the mapping from the intermediate representation to the run-time environment, we introduce the restriction that the intermediate representation should be sum-guarded, which means that operands of the sum operator "}" must be (possibly guarded) action prefix expressions, or sum expressions. This is not a significant restriction, since many specifications (those that are guardedly well-defined) can be transformed into sum-guarded specifications (an algorithm for this is presented in [Kar88]).

The following specification illustrates the intermediate representation (it contains all constructs of intermediate LOTOS, and is sum-guarded):

```
process P[a, b, c](x: integer): noexit :=
  a !x; (let y: integer = x + 1 in P[a, b, c](y))
  []
  [x > 10] ->b ?x: integer; (hide d in P[a, b, d](x) ||[d] Q[d])
  []
  c !x+1; stop
endproc
```

2.3 Run-time environment

We regard implementations to be programs executing in a run-time environment. We describe the run-time environment by specifying the services it provides to programs, and run-time LOTOS is used to describe programs.

We suggest a distributed implementation approach where a LOTOS specification is implemented as a set of communicating tasks. A task is an independent computational unit in a computer system (often the term process is used instead of task, but we use the term task so as not to confuse it with LOTOS processes). Tasks communicate asynchronously by messages sent over a reliable medium, which does not loose or rearrange messages. Each communication involves exactly two tasks, a sender and a receiver. Also, tasks can be dynamically created and removed.

Such an environment can be realised in most computer systems, for example by loosely coupled processors communicating over a network, or within one single process on a traditional operating system, such as a UNIX process.

We assume there exists some mapping from LOTOS data types to data constructs in the run-time environment, and will not further consider the implementation of data types.

To reduce the amount of data operation declarations in subsequent LOTOS specifications, we will use a "declarative" style in data expressions, where we use the choice construct generalised over data values. We do not consider such constructs to be directly implementable, as explained above, but it should be obvious that the same behaviour can be obtained by using a richer set of data operations. We will use generalised choice in two ways: First, as existentially quantified conditions as in the following example:

```
choice e: Element []
  [e IsIn SomeSet] -> g !e; B
```

This should be read as "if there exists some e, such that e is a member of the set SomeSet, g !e is a possible action". Second, choice is used to "receive" values in compound output
"!" experiments, as in:

```
choice arg: Argument []
  g !Constructor(arg); B
```

If some other process can do the action "g !Constructor(value)" where value is an expression representing a data value of sort Argument, the result of a synchronisation between the two processes will be that arg is assigned the value value.

### 2.3.1 Specification of services provided by the run-time environment

We describe the services provided by the run-time environment as LOTOS processes.

Each task is assigned a unique identifier (of sort TaskID) used for addressing the task. A task accesses the medium via an input buffer and an output buffer. A task sends a message by storing it in its output buffer, by actions at gate send, and receives messages by fetching them from the input buffer by actions at gate recv. Actions at these gates have three values: the message, and the identifiers of the source and destination tasks.

Buffers are connected together via a medium, accessed through the gate medium. The behaviour of the medium, and input and output buffers is specified below. Internally, buffers store messages in a Queue data structure. The operation QueueElement constructs a queue element from a message and a task identifier, Head is the first element of a queue, and Tail the rest of the queue. The task parameter is the identifier of the task owning the buffer.

```lotos
process OutputBuffer[medium, send](Q: Queue, task: TaskID): noexit :=
  send ?message: Message !task ?to: TaskID;
  OutputBuffer[medium, send](Append(QueueElement(message, to), Q), task)

choice message: Message, to: TaskID []
[Head(Q) = QueueElement(message, to)] →
  medium !Put !message !task !to;
  OutputBuffer[medium, send](Tail(Q), task)
endproc (* OutputBuffer *)

process InputBuffer[medium, recv](Q: Queue, task: TaskID): noexit :=
  InputBuffer[medium, recv](Append(QueueElement(message, from), Q), task)

choice message: Message, from: TaskID []
[Head(Q) = QueueElement(message, from)] →
  recv !message !from !task;
  InputBuffer[medium, recv](Tail(Q), task)
endproc (* InputBuffer *)
```

```lotos
process Medium[medium]: noexit :=
  medium !Get !message !from !to;
  Medium[medium]
endproc
```
Buffers are composed in parallel with tasks in the following way (assuming $P_{task}$ to be the definition of some task, and $id$ its task identifier):

$$\text{hide send, recv in}
\begin{align*}
\text{InputBuffer[medium, recv]}(\text{EmptyQueue, id}) \\
||\text{recv} \\
\text{OutputBuffer[medium, send]}(\text{EmptyQueue, id}) \\
||\text{send}
\end{align*}
\text{P}_{task}[send, recv, medium](\ldots)
$$

Further, we assume the run-time environment to provide support for creating unique task identifiers. We use a $ID\_Manager$ process for this purpose, defined as follows ($MID$ is a constant, representing the identifier of the $ID\_Manager$, and the operation $Insert$ adds an element to a set):

$$\text{process ID\_Manager[send, recv]}(\text{InUse: SetOfTaskID}: \text{noexit} := \text{recv !RequestID ?from: TaskID !MID; (choice newid: id [] [not(newid IsIn InUse)] → send !ID(newid) !MID !from; \text{ID\_Manager[send, recv]}(id, Insert(newid, InUse)))}}
$$

endproc

A program is executed in the environment in the following way, where $Spec$ is the initial behaviour expression of the program. The operation $\{\}$ is the empty set.

$$\text{Medium[medium]}
\ll
\begin{align*}
&\text{hide send, recv in} \\
&\text{InputBuffer[medium, recv]}(\text{EmptyQueue, id}) \\
&||\text{recv} \\
&\text{OutputBuffer[medium, send]}(\text{EmptyQueue, id}) \\
&||\text{send}
\end{align*}
\text{ID\_Manager[send, recv]}(\text{Insert(MID, \{\})})
\rr

$$\text{hide send, recv in}
\begin{align*}
\text{InputBuffer[medium, recv]}(\text{EmptyQueue, id}) \\
||\text{recv} \\
\text{OutputBuffer[medium, send]}(\text{EmptyQueue, id}) \\
||\text{send}
\end{align*}
\text{Spec[send, recv, medium]}(\ldots)
$$

2.4 Run-time LOTOS

The same set of LOTOS operators as in intermediate LOTOS are allowed in run-time LOTOS, but their usage is restricted in run-time LOTOS.

The three gates $send$, $recv$ and $medium$ are the only gates allowed in run-time LOTOS, and they are used to access buffers as described in Section 2.3.1. $Hide$ statements can be used only to create the gates $send$ and $recv$. 

7
The choice ("[]") operator can be used in only two ways in run-time LOTOS: Either its operands are all (possibly guarded) actions at the gate recv, or its operands are all guarded expressions. In the first case, a choice expression represents a state where a task is prepared to receive different messages. In the second case, choice expressions are used as "case statements". Guarded expressions can also be used in run-time LOTOS as conditional statements.

Parallel composition in run-time LOTOS represents dynamic creation of tasks, and a task deletes itself by the stop operator. It should be obvious how the rest of intermediate LOTOS can be implemented.

3 Transforming intermediate LOTOS into run-time LOTOS

New value identifiers will be created in the transformation. Subsequently, we will assume that the names of these identifiers are unique, so that no naming conflicts arise as a result of the transformation.

For simplicity, we disregard data values in actions, and require actions in summation expressions to be at different gates. In Section 5 we discuss how data values in actions can be implemented.

We describe the transformation from intermediate LOTOS into the run-time environment by a mapping function \(\mathcal{R}\). If \(B\) is an expression in intermediate LOTOS, \(\mathcal{R}(B)\) will be the corresponding expression in run-time LOTOS. The mapping is defined as a set of syntactical translation rules over LOTOS expressions. Expressions written in italics are schematic; lower-case italics represents LOTOS identifiers (terminals), and capitalised italics represents non-terminal LOTOS expressions. For example, the definition

\[
\mathcal{R}(p[\text{Gates}](\text{Values})) = p[\text{gate}, \text{Gates}](\text{Values})
\]

is a rule which defines that the gate identifier gate should be inserted as the first gate parameter in the translation of process instantiations.

3.1 Process definitions and instantiations

Each operand of a parallel operator is mapped to a unique task in the run-time environment. We refer to this kind of tasks as \(P\)-tasks (where \(P\) stands for process). To each \(P\)-task a synchroniser task is associated. A synchroniser negotiates with ports on behalf of its \(P\)-task, in order to establish communication events. Subsequently, the value identifiers (of sort TaskID) \(id\) and \(sid\) are the identifiers of the current \(P\)-task and its synchroniser, respectively. These identifiers are free in some translation rules, but in the generated specification these identifiers will be correctly bound.

Since the use of gates in run-time LOTOS is restricted, gates in intermediate LOTOS cannot be mapped to gates in run-time LOTOS. Instead they will be mapped into value
identifiers of sort GateInfo, which contains the identifier of the port task corresponding
to the gate, and the position of the P-task in the synchronisation tree (the term is due to
Dubuis [Dub89]) of the gate.

A synchronisation tree of a gate shows how behaviour expressions are composed in parallel
with respect to the gate, as shown in Figure 2.

```
Figure 2: Tree structure of hide g in ((P1[g] || P2[g]) || P3[g]) || P4[g] || P5[g]
```

The parallel operator "||" (synchronisation) requires its operand expression to both inter-
act in all events, while the operator "|||" (interleaving) prevents the operand expression
from interacting with each other. The synchronisation tree is traversed to find possible
events at the gate. In the figure, two interactions are possible: either P1, P2, P4, and P5
can interact in an event at gate g, or P3, P4, and P5 can interact.

The position of a P-task in a tree is stored in the GateInfo value parameter corresponding
to the gate. The sort GateInfo is defined below. We use the LOTOS library definition
of the sort String to define a sequence of elements, where <> is the empty string and +
the concatenation operator. The operation GateInfoRecord represents a function that
constructs a value of sort GateInfo from a task identifier and a tree position.

```
type TreePosition is
  sorts Position, ParallelOperation
  opns Synchronisation, Interleaving: -> ParallelOperation
      Left, Right: ParallelOperation -> Position
endtype

(* PositionString is defined as String of Position *)

type GateInfo is PositionString, TaskIdentifier
  sorts GateInfo
  opns GateInfoRecord: TaskID, PositionString: GateInfo
      AppendPosition: Position, GateInfo -> GateInfo
  eqns
      ofsort GateInfo
      forall id: TaskID, pos: Position, positions: PositionString
      AppendPosition(pos, GateInfoRecord(id, positions)) = GateInfoRecord(id, positions + pos)
endtype
```

A process definition is mapped into run-time LOTOS in the following way:

```
R(process p[g1, ..., gm](p1 : sort1, ..., pn : sortn): noexit := B endproc ) =

process p[send, recv, medium](g1, ..., gm: GateInfo,
  id, sid: TaskID,
  p1 : sort1, ..., pn : sortn): noexit :=
```

9
\[ \mathcal{R}(B) \]
endproc

The mapping of process instantiations is analogous:
\[
\mathcal{R}(p[g_1, \ldots, g_m](p_1, \ldots, p_n)) = 
\]
\[
p[\text{send, recv, medium}] (g_1, \ldots, g_m, \text{id, sid, } p_1, \ldots, p_n)
\]

### 3.2 Hiding statements

*Hiding* statements are binding occurrences of gate identifiers, and are mapped into binding occurrences of value identifiers of sort \texttt{GateInfo}. We use the name of the gate as the name of the value identifiers. Also, a port task is created by parallel composition of the Port process definition and the behaviour expression in the hide statement. First, a new task identifier for the port is allocated from \texttt{IDManager}.

\[
\mathcal{R}(\text{hide } g \text{ in } B) = 
\]
\[
\text{send } !\text{RequestID } \text{id !}MID; \\
\text{(choice pid: TaskID [] recv !ID(pid) !}MID \text{id;}} \\
\text{(let g: GateInfo = GateInfoRecord(pid, <> } \text{in (Port[medium](pid) || } \mathcal{R}(B)))}
\]

### 3.3 Parallel composition

The mapping of parallel composition is defined below. \(P(G_S, G_I)\) represents the different variants of the parallel composition operator, where \(G_S\) and \(G_I\) are sequences of identifiers of the gates over which the operands are synchronised and interleaved, respectively. \(G_I\) and \(G_S\) together are all gates that are defined, so if the parallel operator is "||", \(G_I\) is empty, and if the parallel operator is "||", \(G_S\) is empty.

\[
\mathcal{R}(B1 \ P(G_S, G_I) \ B2) = 
\]
\[
\text{send } !\text{RequestID } \text{id !}MID; \\
\text{(choice ptask: TaskID [] recv !ID(ptask) !}MID \text{id;}} \\
\text{send } !\text{RequestID } \text{id !}MID; \\
\text{(choice sid: TaskID [] recv !ID(sid) !}MID \text{id;}} \\
\text{(Synchroniser[medium][sid, ptask] ||} \\
\text{(let id: TaskID = ptask in} \\
\text{hide send, recv in} \\
\text{InputBuffer[medium, recv](EmptyQueue, id)} \\
|| \text{recv])} \\
\text{OutputBuffer[medium, send](EmptyQueue, id)} \\
|| \text{send])} \\
\mathcal{R}_{p_1}(G_S, G_I, B1))))) \\
|| \\
\text{send } !\text{RequestID } \text{id !}MID; \\
\text{(choice ptask: TaskID [] recv !ID(ptask) !}MID \text{id;}} \\
\text{send } !\text{RequestID } \text{id !}MID; \\
\text{(choice sid: TaskID [] recv !ID(sid) !}MID \text{id;}}
(Synchroniser[medium](sid, ptask))
||
(let id: TaskID = ptask in
  hide send, recv in
  InputBuffer[medium, recv](EmptyQueue, id) || [recv]
  OutputBuffer[medium, send](EmptyQueue, id) || [send]
  R_{right}^P(G_S, G_I, B2)))))

The translation functions \( R_{left}^P \) and \( R_{right}^P \) re-binds GateInfo identifiers by appending
a tree position to their values. Expressions such as "\( g \cdot G \)" denotes a sequence of gate
identifiers. The first element in the sequence is \( g \) and \( G \) is the rest of the sequence. The
symbol \( \epsilon \) is the empty sequence. We show only the definition of \( R_{left} \), since the definition
of \( R_{right} \) is analogous.

\[
R_{left}^P(g \cdot G'_S, G'_I, B) = \\
\text{let } g: \text{GateInfo} = \text{AppendPosition(Left(Synchronisation), g) in } R_{left}^P(G'_S, G'_I, B)
\]

\[
R_{left}^P(\epsilon, g \cdot G'_I, B) = \\
\text{let } g: \text{GateInfo} = \text{AppendPosition(Left(Interleaving), g) in } R_{left}^P(\epsilon, G'_I, B)
\]

\[
R_{left}^P(\epsilon, \epsilon, B) = R(B)
\]

### 3.4 Summation expressions

A summation expression in intermediate LOTOS represents a state where a process is
prepared to participate in a communication event (since operands of the "[]" operator are
in intermediate LOTOS required to be action prefix expressions). In run-time LOTOS
this is represented by the P-task sending a request to its synchroniser to participate in an
event (RequestEvent). The parameter of the request is a set of values of sort GateInfo
representing the different gates in the summation expression. We use the LOTOS library
definition of Set to construct sets of values, where Insert adds an element to a set and
{} is the empty set.

The synchroniser will negotiate with the corresponding port tasks, which may result in
the P-task being selected to participate in an event. If this happens, the synchroniser
notifies the P-task of this by sending an ConfirmEvent message, which has a parameter
that identifies the selected event.

\[
R(g_1; B_1[])\ldots[g_n; B_n) = \\
\text{send } !\text{RequestEvent(Insert(g_1, \ldots, Insert(g_n, \{\}) \ldots)) } \text { lid \ lid;} \\
\text{(choice gate: GateInfo} \\
\[ \\
\text{recv } !\text{ConfirmEvent(gate) } \text{laid;} \\
\text{(|gate = g_1) } \rightarrow R(B_1) \\
\]

11
\[ \text{\[\text{\{gate = } g_n \text{\} } \rightarrow \mathcal{R}(B_n)\]\}]

3.5 Let and guarded expressions

Both let and guarded expressions are mapped unmodified into run-time LOTOS.

4 Implementing synchronisation

In this section, we informally describe the protocol by which synchroniser tasks and port tasks negotiate in order to establish communication events. A formal specification is given in Appendix A. As described earlier, a P-task requests to interact by sending a RequestEvent to its associated synchroniser. As a result, the synchroniser will send Notify messages to the ports representing the gates at which the P-task is prepared to interact.

When a port has received Notify messages enough to deduce that some P-tasks can interact, it starts negotiating with the corresponding synchronisers. The port requests the synchronisers, one by one, to commit themselves to an interaction at the port, by sending a Lock request to a synchroniser and then awaiting a response. If a synchroniser that receives a Lock request is prepared to commit itself to the interaction, the synchroniser responds by sending a Yes message to the port, and the port will then try to lock another synchroniser. When all synchronisers involved have responded with Yes messages, the interaction has been established. The port acknowledges this by sending Success messages to all the synchronisers. In turn, those synchronisers send ConfirmEvent messages to their P-tasks.

A synchroniser can also respond with a No message to a Lock request, if some other port already has established an interaction involving the P-task concerned. If the port receives a No message, it "forgets" everything about that P-task and will not try to commit the synchroniser to an interaction again (unless it receives another Notify message from the synchroniser).

When a port receives a No message, it means that the port has failed to establish an interaction. If some other synchronisers already have committed themselves to the interaction (that is, have responded with Yes messages) the port must relieve those synchronisers from their commitment. To do this, the port sends Abort messages to the synchronisers which thereafter are free to commit themselves to any port.

It is possible that a synchroniser receives a Lock request from a port while it is already committed to some other port. There are several possible solutions for this situation. We choose to not let the synchroniser respond to the Lock request until it knows whether the port to which it is committed has managed to establish an interaction. That is, the synchroniser defers the response to the Lock request until it receives Success or Abort from the port to which it is committed. This approach is the most common in similar
protocols, and we prefer this approach since it requires less messages to be sent between tasks.

Using this approach there are potential deadlocks. For example, assume that the two synchronisers $Sync_i$ and $Sync_j$ have requested their P-tasks to communicate at ports $port_p$ and $port_q$. $Port_p$ sends a Lock request to $Sync_i$ which replies with a Yes and is thereby committed to $port_p$. At the same time, $port_q$ locks $Sync_j$ in the same way. Thereafter, $port_p$ sends a Lock request to $Sync_j$, and $port_q$ to $Sync_i$. Both requests will be deferred since both $Sync_i$ and $Sync_j$ already are committed, and a deadlock has occurred.

To prevent this situation we require synchroniser identifiers to be totally ordered, which means that there exists some total ordering relation $\leq$ on identifiers, that is, for each pair of different identifiers $i$ and $j$, either $i \leq j$ or $j \leq i$ hold. A port sends Lock request to synchronisers in order. In the example, if $i \leq j$, both $port_p$ and $port_q$ will first send the Lock request to $Sync_i$. Only one of $port_p$ and $port_q$ will receive a Yes response (since the other response will be deferred), and can proceed and send a Lock message to $Sync_j$.

### 4.1 An example

In figure 3 a timing diagram is shown for communication between one P-task and its associated synchroniser, and two ports (port 1 and port 2). First the P-task announces

![Figure 3: Timing diagram for one P-task and two ports](image)

its willingness to interact at the two ports by sending a RequestEvent message to the synchroniser. The synchroniser sends a Notify message to the ports, to inform the ports that the P-task is now ready to interact. Port 2 then discovers that an interaction is possible, and tries to commit the synchroniser by sending a Lock request. Since the synchroniser has not committed itself to any other port, it immediately responds with a Yes.

Thereafter, port 1 also tries to commit the synchroniser by a Lock request. The synchroniser is already committed to port 2, and therefore defers its answer to port 1. However, port 2 fails in committing all synchronisers required to establish an event (that is, some synchroniser answers with a No to port 2’s Lock request). Port 2 must therefore release the synchroniser by an Abort message.
The synchroniser will then immediately send the (deferred) response to port 1's Lock request, and in this case, the response will be positive (Yes). Port 1 manages to commit all synchronisers required, and announces the successful negotiation with a Success message to the synchroniser. The synchroniser will in turn inform the P-task that it has been appointed to participate in an event, by sending a ConfirmEvent message.

5 Interactions with data values

Since we have disallowed value generating interactions (where all participants offer to do an input ("?")) experiment), the inclusion of data values in interactions is straightforward: The information sent from a P-task to its synchroniser, in RequestEvent messages, is extended to include information about values in actions. This information is passed on to the ports, in Lock messages, which use the information to compute what the possible events are. For input experiments, a port computes the corresponding values and transfers them to the P-task.

A selection predicate imposes restrictions on values exchanged in interactions. For example, the predicate in the action g ?x: integer [x > 17] requires the received value to be greater than 17. The implementation of selection predicates depends on how data types are implemented. The most efficient, in terms of number of messages sent, would be to evaluate selection predicates in ports: A P-task sends the selection predicates unevaluated to the ports, so the implementation of data types must support "lazy" evaluation of predicates. The port computes possible events by evaluating the selection predicates with respect to output ("!") values proposed by other P-tasks. Since a selection predicate might contain references to data variables in the P-task, this requires local data of the P-task to be accessible by the port.

It is also possible to instead evaluate predicates in P-tasks: The port suggests a value to the P-task (as a parameter of the Lock requests), and the P-task evaluates the guard with respect to the value. If the predicate holds for the value the P-task acknowledges the request, otherwise the request is rejected. This seems to be easier to implement, but might require more messages to be sent then the previous approach, since a port may have to try with several Lock requests before the P-task accepts the request.

6 Related work

In this section, we summarise some approaches presented for implementing LOTOS specifications. The HIPPO toolset, developed within the SEDOS project [vE89], and the LOTOS interpreter from the University of Ottawa [LOBF88] are tools developed for simulating LOTOS specifications, and provide support for examining the dynamic behaviour of a specification. In these tools, behaviour expressions are executed by a LOTOS simulator, or interpreter, based on the axioms and inference rules. This simulator performs a symbolic execution of a specification. The axioms and inference rules are used to compute the actions, and the resulting behaviour expressions, of a behaviour expression. One event
is chosen by the user (or possibly by the simulator itself), and the procedure is repeated for the resulting behaviour expression.

In [vBGW89], an implementation approach is presented where a behaviour expression is represented by a tree structure in the run-time environment (called activity tree) where the nodes in the tree represents LOTOS operators. The leaf nodes represent the actions of the behaviour expression. Gates are represented by routes, called virtual rings, through leaf nodes in activation trees, and negotiation of events is performed by sending messages through virtual rings. When a communication event has been established the tree is updated to reflect the new state of the system.

In [Gil87], it is shown how LOTOS constructs can be compiled into constructs in PARLOG, a concurrent logic programming language. Due to the similarities between the two languages, many LOTOS constructs can be directly mapped into PARLOG constructs. The process communication models of the two languages are, however, quite different so additional PARLOG processes are required to implement LOTOS events.

An implementation approach based on the fact that a LOTOS specification is a definition of a transition system is presented in [Kar88]. An algorithm is presented for transforming LOTOS specifications into an extended finite transition system representation. The target language is Estelle [IS9], which is a formal description technique, based on transition systems extended with data variables. In this approach, a LOTOS specification is translated into a monolithic transition system. In [FQVM88] and in [Dub89], it is instead suggested that specifications can be translated into a set of communicating extended transition systems. In the former, LOTOS processes are implemented as MODULA 2 modules, and a "common memory based protocol" utilising a "two phase locking algorithm" is used for establishing events, but the protocol is not described in more detail. In the latter, an algorithm is given for translating LOTOS specifications into "automatons" communicating via ports, an approach similar to what has been presented in this paper.

7 Summary

We have shown how a subset of LOTOS specifications can be translated into implementations. The most significant characteristics of this subset is that internal events "i" and choices ranging over value domains (such as value generating interactions) are not allowed since, in our opinion, they are abstractions that need to be refined to be directly implementable.

Specifications are translated into implementations in environments providing process communication primitives based on message passing. We have chosen this implementation environment since it is general, that is, can be realized in most computer systems, and allows for implementations to exploit parallelism in parallel computer systems to gain performance. We have modeled implementations by specifying the services provided by an asynchronous environment, and by defining a LOTOS subset, called run-time LOTOS, which is used to describe programs executing in such environments. We have presented a transformation from intermediate LOTOS to run-time LOTOS.
Parallel composition is implemented by creating independent processes, call P-tasks, that run in parallel. Gates are implemented by processes called port tasks. In order to establish communication events, port tasks and P-tasks communicate according to a protocol. The protocol we have presented is based on protocols for synchronising CSP processes.

We have not attempted to prove the protocol correct, nor to formally show that the implementation obtained by the transformation is consistent with the original specification. These are directions of future work. To refine the algorithm, and study its utility, practical experiments are required.

Acknowledgements

I am grateful to Fredrik Orava and Joachim Parrow for valuable comments and criticism. I also would like to thank Eric Dubuis for stimulating discussions.

References


ISO. *ISO Standard 9074: Information processing systems—Open systems interconnection—Estelle—A formal description technique based on an extended state transition model.*


### A Formal specification of the protocol

(*
   Synchroniser—allocate I/O buffer, then instantiate Sync process definition.
*)

```plaintext
process Synchroniser [medium] (sid, id: TaskID): noexit :=
    hide send, recv in
    InputBuffer[medium, recv](EmptyQueue, sid)
[recv]
    OutputBuffer[medium, send](EmptyQueue, sid)
[send]
    Sync[send, recv](sid, id)
```

where

(*
   Sync—Reject all Lock requests received. When a RequestEvent message is received from synchroniser, create Negotiator and Notifier processes to execute in parallel
)
*)
process Sync[send, recv](sid, id: TaskID)
(choice pos: PositionString
  []
    recv !Lock(pos) ?p: PortID !sid;
    send !No !sid !p; Sync [send, recv] (id, sid)
  )
(choice actions: GateInfoSet
  []
    recv !RequestEvent(actions) !id !sid;
    Negotiator [send, recv] (sid, id, actions)
    ||
    Notifier [send, recv] (sid, id, actions)
  >>
    Sync [send, recv] (sid, id)
  )
)

where
(*
  Notifier—send a Notify message to all ports present in RequestEvent message, then terminate.
  *)
process Notifier [send, recv] (sid, id: TaskID, not_notified: GateInfoSet): exit :=
(choice p: TaskID, pos: TreePositionString, e: Experiment
  []
    [GateInfoRecord(p, pos) isIn not_notified] →
    send !Notify(pos) !p !sid;
    Notifier[send, recv][sid, id, Remove(GateInfoRecord(p, pos), not_notified)])
  ]
  [not_notified eq {} of GateInfoSet] → exit
endproc (* Notifier *)

(*
  Negotiator—accept valid Lock requests and then instantiate Committed process definition. Invalid Lock requests are rejected.
  *)
process Negotiator [send, recv] (sid, id: TaskID, actions: GateInfoSet): exit :=
choice pos: PositionString
  []
    recv !Lock(pos) ?p: TaskID !sid [GateInfoRecord(p, pos) isIn actions];
    send !Yes !sid !p;
    Committed [send, recv] (sid, id, p, actions, {} of TaskIDSet)
  []
    recv !Lock(pos) ?p: TaskID !sid [not(GateInfoRecord(p, pos) isIn actions)];
    send !No !sid !p;
    Negotiator [send, recv] (sid, id, actions)
)

where
(*
  Committed—one Lock request has been accepted, wait for the event to be confirmed or aborted. Incoming valid requests are not answered until later, invalid requests are rejected.
  *)
process Committed [send, recv] (sid, id: TaskID,
p: TaskID,
actionoffers: GateInfoSet,
delayed: TaskIDSet): exit :=

choice pos: PositionString

[]
recv !Lock(pos) ?p1: TaskID !sid [GateInfoRecord(p1, pos) IsIn actions];
Committed [send, recv] (sid, id, p, actions, Insert(p1, delayed))
[]
recv !Lock(pos) ?p1: TaskID !sid [not(GateInfoRecord(p1, pos) IsIn actions)];
send !No !sid !p1;
Committed [send, recv] (sid, id, p, actions, delayed)
[]
recv !Confirm(pos) !p !sid;
Succeeded [send, recv] (sid, id, delayed, GateInfoRecord(p, pos))
[]
recv !Abort !p !sid;
([delayed eq {} of PortIDSet] →
  Negotiator [send, recv] (sid, id, actions)
[]
  (choice p1: TaskID
    []
    [p1 IsIn delayed] →
    send !Yes !sid !p1;
    Committed [send, recv] (sid, id, p1, actions, Remove(p1, delayed))
  )
)

where

(*
  Succeeded—an event as been established. Reject all delayed Lock requests (CleanUp)
  and then confirm the event to P-task and terminate.
*)
process Succeeded [send, recv] (sid, id: TaskID,
delayed: PortIDSet,
  event: GateInfo): exit :=
CleanUp [send, recv] (sid, id, delayed) >> send !ConfirmEvent(event) !sid !id; exit

where

(*
  CleanUp—answer No to all delayed Lock requests.
*)
process CleanUp [send, recv] (sid, id: TaskID, delayed: TaskIDSet): exit :=
[delayed eq {} of TaskIDSet] → exit
[]
  (choice p: TaskID
    []
    [p IsIn delayed] →
    send !No !sid !p; CleanUp [send, recv] (sid, id, Remove(p, delayed))
  )
endproc (* CleanUp *)
endproc (* Succeeded *)
endproc (* Committed *)
endproc (* Negotiator *)
endproc (* Sync *)
endproc (* Synchroniser *)
19
TaskPosition data type—internal port representation of action offers. Values of sort TaskPosition consists of a synchroniser identifier and a position in a synchronisation tree, and are constructed by the operation TaskPositionRecord. PossibleEvent and LeastID are functions over a set of values of sort TaskPosition. The value of PossibleEvent is the subset of TaskPosition values that can be combined into one event. The value of LeastID is the member of the set that has the least synchroniser identifier.

TaskPositionSet is defined as Set of TaskPosition.

type TaskPositions Is TaskPositionSet
opns TaskPositionRecord: TaskID, Position → TaskPosition
   PossibleEvent: TaskPositionSet → TaskPositionSet
   LeastID: TaskPositionSet → TaskPosition
endtype

Port process—Create a Locker and a Recorder process to run in parallel. First allocate IO-buffers for Recorder.

process Port [medium] (id: TaskID): noexit :=
   hide send, recv in
      InputBuffer[medium, recv](EmptyQueue, id)
      [recv]
      OutputBuffer[medium, send](EmptyQueue, id)
      [send]
      send !RequestID lid !MID;
      (choice rid: TaskID []
         recv !ID(rid) !MID lid;
         ( Recorder [send, recv] (rid, id, {} of TaskPositionSet)
            ||
            Locker [send, recv] (id, rid)))

where
   Recorder—Store data from incoming Notify messages. When some P-tasks can be combined into one event, send the corresponding set of TaskPosition values to Locker process (in a Try message). If the Locker process fails to establish an event, it will send the values back to Recorder (in a Failed message).

process Recorder [send, recv] (rid, id: TaskID, requests: TaskPositionSet): noexit :=
   (choice pos: PositionString
       recv !Notify(pos) lid ?sid: TaskID;
       Recorder [send, recv] (rid, id, Insert(TaskPositionRecord(sid, pos), requests))
   )
   []
   [not(PossibleEvent(requests) = {}) →
      send !Try(PossibleEvent(requests)) !rid lid;
      Recorder [send, recv] (rid, id, requests Minus PossibleEvent(requests))
   ]
   (choice failed: TaskPositionSet [])
recv !Failed(failed) lid !rid;
   Recorder [send, recv] (id, id, failed Union requests))
)
endproc (* Recorder *)

(*
   Locker—Wait for a Try message from Recorder, then instantiate Lock process.
*)

process Locker [send, recv] (id, rid: TaskID): noexit :=
choice matching_requests: TaskPositionSet []
   recv !Try(matching_requests) !rid lid;
   Lock [send, recv] (id, rid, matching_requests, {} of TaskPositionSet)
where
(*
   Lock—Try to commit a set of Synchronisers to an event. Instantiate process definition
   Confirmer if the attempt succeeds, otherwise instantiate process definition Aborter.
*)

process Lock [send, recv] (id, rid: TaskID,
   not_locked, locked: TaskPositionSet): noexit :=
   [not_locked eq {} of TaskPositionSet] →
   (Confirmer [send, recv] (id, rid, locked) >> Locker [send, recv] (id, rid))[
   (choice sid: TaskID, pos: PositionString
      []
         [TaskPositionRecord(sid, pos) isIn not_locked and (sid = LeastID(not_locked))] →
            send !Lock(pos) lid !sid;
            (recv !Yes !sid lid;
               Lock [send, recv] (id, rid
               Remove(TaskPositionRecord(sid, pos), not_locked),
               Insert(TaskPositionRecord(sid, pos), locked))
         )
   ]
)
where
(*
   Aborter—instantiated if port fails to establish an event. Release any committed syn-
   chroniser from their commitment, by Abort messages.
*)

process Aborter [send, recv] (id, rid: TaskID, locked: TaskPositionSet): exit :=
   [locked eq {} of TaskPositionSet] → exit
   []
   (choice sid: TaskID, pos: PositionString
      []
         [TaskPositionRecord(sid, pos) isIn locked] →
            send !Abort !sid !rid;
   )
Aborter [send, recv] (id, rid, Remove(TaskPositionRecord(sid, pos), locked))
)
endproc (* Aborter *)

(*
   Confirmer—instantiated if port succeeds in establishing an event. Confirm the event
to all synchronisers involved, by Confirm messages.
*)

process Confirmer [send, recv] (id, rid: TaskID, locked: TaskPositionSet): exit :=
  [locked eq {} of TaskPositionSet] → exit
[]
(choice sid: TaskID, pos: PositionString
  []
  [TaskPositionRecord(sid, pos) IsIn locked] →
    send !Confirm(pos) lid !sid;
    Confirmer [send, recv] (id, rid, Remove(TaskPosition(sid, pos), locked))
)
endproc (* Confirmer *)
endproc (* Lock *)
endproc (* Locker *)
endproc (* Port *)