Chapter 7

LOTOS and Process Algebras

7.1 Basic Concepts

The LOTOS specification language is an acronym standing for the Language of Temporal Ordering Specifications, but it is not a temporal logic. Instead it belongs to a family of languages known as process algebras. In such languages, the process is the basic element of the language. Most components in the system are viewed as processes, even some that normally would not be considered processes in programming languages. A crucial aspect of the approach is that there are systematic, compositional ways to build more complex processes out of simpler ones. The basic mathematical tool is simple function composition, but parallelism is more complex in its definition. Sequences of actions can be grouped by using regular parentheses. The semantics of a process is known as a behavior and is based on possible sequences of actions or events, rather than on states. A black-box approach is taken, where only those actions visible to the environment are significant in defining the semantics.

Other well known languages in this family include CSP and CCS. The original purpose of process algebras was to consider fundamental questions on the nature of concurrency, and the differences among various models for concurrency. Thus special attention has been paid
to questions of when processes can be considered equivalent, and how possibilities for deadlock affect the meaning of a process.

The main application for these languages in specification of concrete systems is in treating the control aspects —communication and synchronization— of a collection of distributed interacting processes. On the data level, LOTOS includes the ACT-ONE notation for algebraic specifications, equivalent to a restricted version of the Larch Shared Language. Basic data types and operations are defined in the algebraic component and then used in the process definitions and specifications of LOTOS. The language may therefore be seen as an extremely well developed interface language in the Larch style, designed to handle processes and concurrency, and especially communication protocols.

A LOTOS specification describes legal behaviors as processes with gates. The gates provide the interfaces among processes and with the environment, and determine which actions will be visible. As noted above, a behavior is an expression that describes legal sequences of the visible actions, organized into trees or transition graphs. A typical process (behavior) is to take some action, followed by another behavior, i.e., by a process. A behavior looks like a collection of recursive function definitions, using tail recursion since the continuation never ‘returns’ to the original behavior being described. A simple behavior $A$ can use the semicolon symbol between an action and a behavior ($a;B$), meaning that action $a$ is followed by behavior $B$. There is an associated graphic notation that represents an action as a labelled edge and a behavior as a triangle, so that the process $A$ can be described as an edge labelled $a$ connected to a triangle labelled $B$.

Many LOTOS specifications involve nondeterminism. Some other notations, such as statecharts and the Statemate system, view nondeterminism as a potential error in the specification to be pointed out to the user. In LOTOS, nondeterminism is viewed as a useful tool. It allows hiding lower level implementation decisions that are irrelevant, expressing unknown relative speeds of messages on channels, or modeling other unknown aspects of the system. A choice symbol between behaviors ($A \sqcup B$) means the behavior of either $A$ or $B$, and corresponds to a nondeterministic choice. Graphically, such an expression is drawn as two triangles joined at one vertex, so the new root has the union of steps possible from the two original roots. Note that there is
no separate step representing the nondeterministic choice, so the new tree merely identifies together the original roots.

7.2  More on Processes and Gates

A process interacts with its environment through the gates it declares. The only actions of the process that are observable are those involving the gates. One such action is to offer a value at a gate. This is denoted in an expression of the form `gate-name ! expression`. The exclamation mark is used here as an operator, and not as a suffix to a name as was seen in Z. Nevertheless, it still has a similar meaning to the output variables of Z, since this will correspond to output of a value at the gate.

Similarly, an action of the form `gate-name ? variable` expresses that the process will accept a value from the environment through the named gate, storing it in the variable. This action therefore corresponds to input. Later we will see that offering and accepting values can be generalized beyond simple input/output of processes.

For any action (either with ! or with ?) involving a gate that is visible to the environment, that action can only be executed if the environment is also able to execute an action involving that gate. Otherwise the action cannot be chosen to execute. If there are alternative possibilities (in a nondeterministic choice), one of them can be chosen. Otherwise the process cannot proceed until the environment is willing to also execute an action involving the gate.

A process with gates `ask` and `answer` could have in it

\[
(\text{ask} \ ? \ x \ ; \ \text{answer} \ ! \ (x + 1) \ ) \ [] \ \text{answer} \ ! \ 0
\]

This means that either the process can accept a value for \(x\) from the environment at the `ask` gate and then offer the environment the value accepted plus one at the `answer` gate, or, alternatively, the process may immediately offer the environment the value 0 at the `answer` gate.

An action can also be preceded by a guard, written as a predicate within brackets followed by an arrow to the action. The predicate must be true in order for the following action to be the selected behavior. Guards are most usually used in a choice action, to restrict the choice
when the guard does not hold. However, there is no need for the guards
to be mutually exclusive, i.e., the choice can still be nondeterministic
after all guards have been evaluated. We might have:

\[
( \text{[ nonempty(queue) = true ]} \rightarrow \text{give } ! \text{ head(queue)}) \quad \Box \\
(\text{[ nonfull(queue) = true ]} \rightarrow \text{put } ? \text{ temp })
\]

The operators head, nonempty, and nonfull are assumed to come
from an appropriate algebraic definition of a queue. If the queue is
nonempty one possible behavior is to offer the value of the head of
the queue at the give gate, and if the queue is not full, a possible
behavior is to accept a value for the local variable temp at the put
gate. If both conditions are true, the choice among the behaviors is
nondeterministic as far as the process is concerned, and may depend
on whether the environment is willing to execute an action involving
the put or the give gate, or both.

Another language feature with a similar purpose to guards is known
as a selection predicate. This is also a bracketed predicate, but now
appearing immediately after a gate synchronization statement, and in-
cluding restrictions on the values offered or accepted in the statement.
It can be seen as looking ahead to evaluate whether the gate synchro-
nization involves acceptable values (from the point of view of the pro-
cess containing the predicate). If not, the communication cannot occur.
That is, the predicate is guaranteed to be satisfied by any synchroniza-
tion and assignment of values that is finally chosen.

To express that a process may have additional internal structure
and operations that are irrelevant to the actions visible at the declared
gates, the letter i is used. When i appears as an action, it means
that some internal operation is done, without specifying the nature of
that operation. This gives an approximation of the statechart zoom
capability. If the process is examined internally, it may have relevant
operations, but at the level being considered, those operations are hid-
den, with only the i left to indicate that other operations may be going
on.

LOTOS specifications are textual, but can be built up very similarly
to statecharts, as a collection of lower level processes, some of which
 correspond to processes as normally understood, and some of which
correspond to independent components of the state. As in statecharts
and in Z, it is possible to hide internal actions of a process. Such hiding
can be useful when the internal actions have not yet been fully specified.
Even when the actions already have been specified, hiding can be used
to allow concentrating on high level interactions at some gates, without
considering the details of internal activities of the processes at the same
time.

In particular gates can be hidden explicitly in LOTOS by

\[ \text{hide } \langle \text{gate-list} \rangle \text{ in } \langle \text{behavior} \rangle \]

This means that a behavior is defined in which the actions involving the
gates in the list are replaced by \( i \), those gates are no longer visible, and
as will be seen below, will no longer synchronize with external events
involving the same gate names. For a LOTOS fragment

\[ \text{hide ask in } ( ( \text{ask } ? \text{x}; \text{answer } ! (\text{x }+\text{1})) \[ \text{answer } ! \text{0} ) \]

the behavior is equivalent to

\[ ( i ; \text{answer } ! (\text{x }+\text{1}) ) \[ \text{answer } ! \text{0} ) \]

The process can either do some internal action and offer \( x + 1 \) at the
answer gate, or immediately offer 0 at that gate. Although the ask
gate and any action at that gate is now hidden, the fact that an internal
action is possible as one option may be important. For example, if
the internal action were not present in the above expression, the envi-
ronment could affect which of the offerings of values at the answer gate
could be selected, in effect controlling which choice would be made. In
particular, the environment could be at a statement answer ? \( y \) [ \( y \]
= 0 ] which may only be consistent with the second option above.
With the internal action \( i \) as the first step in one of the nondetermi-

nistic choices, a process can decide independently of the environment to
choose the internal action, and then only offer the \( x + 1 \) option to the
environment.

Another important built-in action is stop. This action denotes
an absolute black box: no behavior can be observed if this action is
executed. The stop action can be used to indicate explicitly that a
system can be in a deadlock, or can completely terminate. If one option
in a choice statement is stop (or is equivalent to stop), that option is in effect closed as if it had a false guard. A stop action can only be chosen when there is no immediate alternative. If the internal operation in the expression above could be shown to loop forever with no visible action or to itself deadlock, it could be replaced by stop. Then the option that chooses the left behavior in the choice statement is closed, and the system could only offer 0 at the answer gate.

Another built-in action is written as exit. Like stop, execution of the exit action terminates the process instance in which it is contained. However, following an exit action a process may be composed with another process in ways that allow continuing the second process. This is not true for a stop action. An exit behavior represents proper termination. The ways in which processes can be combined using exit will be considered after process definitions and instantiations are explained.

### 7.3 Process definitions and instances

Another crucial aspect of LOTOS specifications is that ‘abstract’ or ‘formal’ processes can be defined with formal parameters, and then various instances of those can be declared, with different parameters, including gate names. This allows reuse of similar forms in a variety of contexts. Processes can be declared once, and then instantiated often. In effect, a type of process is defined, and then multiple instances, each with its own gates and parameters, can be derived from it.

A process definition consists of a header, a behavior, and local definitions. The header contains a process name, formal gate names, formal variable parameters, and what is called the functionality. The behavior uses LOTOS expressions in terms of the formal parameters and gates. The optional local definitions can define terminology used in the behavior section using algebraic specifications, or include locally defined auxiliary processes. Syntactically, the form of a process definition is:

```
PROCESS
    proc-name [ gate-names ] (form-params): functionality :=
    behavior
WHERE
    local-definitions
```
The functionality defines a list of the types of values that can be passed to another process after the defined one properly terminates. If the process definition is not intended to terminate, and has no exit behaviors, its functionality is the keyword noexit. If it is intended to properly terminate, but does not pass on values, the functionality is exit. Otherwise, the functionality of the process is the keyword exit followed by a sequence of data types in parentheses. All behaviors of an instance of the defined process should determine values for the expressions that must appear after every appearance of exit within the process definition. The values must match the types appearing in the functionality.

Within the local definitions (or globally in a library), the data aspects of a system can be handled exactly as in Larch, using the ACT-ONE algebraic notation that is part of LOTOS. The differences between ACT-ONE and Larch are largely syntactic, and the underlying principles are the same. However, here only the basic abstract data operations are handled in the algebraic notation, and the control aspects are treated in the rest of the LOTOS process algebra. Thus synchronization messages between the components of the system are not directly treated in ACT-ONE, since they can be conveniently expressed in LOTOS.

To illustrate some of the above ideas, consider a process definition that models an unreliable communication line that can sometimes throw away a bit sent along it. Of course, this would not be part of a desired specification, but could describe some problematic hardware, that would be corrected by a higher level communication protocol that somehow overcomes the difficulty by resending (e.g., using an alternating bit protocol). We can define

```plaintext
process BADLINE [ Get, Put ] : NOEXIT :=
   Get ? bit : bool
; ( i
   ; Put ! bit
   ; BADLINE [Get, Put] )
[]
( i
```
This process definition has two parametric gates, and is declared to be nonterminating (the keyword \texttt{NOEXIT} is its functionality). It will receive an input on line \texttt{Get} and then can either do some internal step followed by sending the bit received on \texttt{Put}, and begin again, or do some possibly different internal step, not send anything, and repeat from the beginning.

A similar definition could be made to model a faulty line that reversed bits occasionally, by inserting \texttt{Put \ texttt{not(bit)}} after \texttt{i} in the second alternative.

The process definition \texttt{BADLINE} can be instantiated several times with different gates, that in effect define different versions of the faulty channel. Thus \texttt{BADLINE \ [ frrear, tofront \] } could connect a component that has a gate \texttt{frrear} to one with a gate \texttt{tofront}, while other instances of \texttt{BADLINE} are independent channels with no connection to this one, except their mode of operation.

\section{Parallelism and synchronization}

The environment of a process is most often a collection of other processes with which it can be composed in parallel.

The simplest form of parallelism does not involve synchronization or communication and is denoted by \texttt{A || B}. A behavior is defined in which the actions of \texttt{A} and \texttt{B} are interleaved. That is, at each step either a possible action of \texttt{A} is done, or a possible action of \texttt{B}. Even if the same gate names appear in both processes, each action can be done without regard for actions in the other process. The visible gates are simply the union of the visible gates of the two processes. A process modeling a bidirectional faulty line could be defined by using two interleaved instances of the previously defined unreliable line:

\begin{verbatim}
process bidirect[In1, Out1, In2, Out2]: noexit :=
BADLINE[In1, Out1] ||| BADLINE[In2, Out2]
endproc
\end{verbatim}
More generally, parallel processes can be synchronized at some gates in what is known as *partial synchronization*. This is denoted by

\[
A \mid \langle \text{gatelist} \rangle \mid B
\]

and means that events involving the gates in the list must synchronize between \(A\) and \(B\) with a synchronization event on that gate, as described below.

The most common form of synchronization event involves a value offering in one process on a gate on which synchronization is desired, and a value acceptance in the other process. In this case the offering corresponds to output, and the acceptance corresponds to input. We have seen that offering a value is denoted by an exclamation point between the name of the gate and the expression offered, and accepting a value is denoted by a question mark between the name of the gate and a variable that can accept the value. If \(A\) has as its possible next action \(G \! \! \! .5\) and \(B\) has a possible \(G \! \! ? x\), where \(G\) is in the list of gates on which they synchronize, then both events (simultaneously and atomically) can occur as a synchronization event. The value offered is associated with the accepting variable. This can be seen as equivalent to an assignment statement of the offered value to the accepting variable.

To integrate the possibly faulty bidirectional communication line described above with a process \textbf{Front} that has a channel on which it receives data bits and another on which it sends, and a process \textbf{Back} organized similarly, we could have:

\[
(\text{Front}[\text{fout}, \text{fin}]) \mid (\text{fout}, \text{fin}) \mid \text{bidirect}[\text{fout}, \text{bin}, \text{bout}, \text{fin}] \mid (\text{bin}, \text{bout}) \mid \text{Back}[\text{bin}, \text{bout}]
\]

This means that there will be a faulty line in each direction, where \textbf{Front} interacts with the gate \textbf{fout} where it sends its output for later transmission to the back end, and the gate \textbf{fin} where it receives values from the channel, after they were earlier transmitted from the back end. The processor \textbf{Back} receives its data from the gate \textbf{bin} and sends its results along gate \textbf{bout}.

It is possible to have two processes at actions involving the same gate, where both are offering values. A synchronization event can occur if the values offered are the same. No value is transferred, but the joint
action still can act as a synchronization point between the processes. Similarly, a joint action can involve two value acceptances, say \( G \ ? \ x \) and \( G ? \ y \). In this case, the processes can synchronize, agree on any value from the domain, and assign that value to \( x \) and to \( y \).

A series of value offerings and/or acceptances is also possible at each gate, where each item in the series must have a consistent value that may be chosen. The behavior

\[
(a ! 5 \ ? \ x ) ([a]) (a \ ? \ y \ ? \ w)
\]

is equivalent to a single action where the right process accepts the value 5 in \( y \), and both processes accept the same arbitrary value from the domain in \( x \) and \( w \), respectively, assuming that all types agree. So a synchronization can succeed if the final value is agreed among the participating processes, for each item in the series of offerings or acceptances.

If the synchronization among the processes above on gate \( a \) is intended to be an internal event of the behavior, with no relation to the environment, then the phrase \texttt{hide a in} \ldots can be written before the behavior when it is embedded in more complex contexts.

If this is not done, partial synchronization on a gate does not hide that gate. In other words, two processes that synchronize on a gate can themselves define a behavior that synchronizes with a third process on the same gate. Thus a synchronization event is not restricted to just two participant processes. In such as behavior all three offerings and/or acceptances must be consistent for the synchronization action to occur. Even more complex behaviors can be constructed to form multi-party events with any desired number of participant processes and subprocesses, as long as the outcome is consistent. This provides a continuum of synchronization conditions that resemble unification conditions among logical predicates in languages such as Prolog. The binding that occurs gives the effect of input/output or of an assignment statement if offerings and acceptances are matched.

Consider a bank system with a collection of transactions. If there is a process \texttt{Sourcebank[T]} where \( T \) is a gate on which funds can be transferred out, while \texttt{Targetbank[S]} is a process that can accept funds on gate \( S \), then renaming and partial synchronization can be used to model an instance of transferring funds as
7.4. PARALLELISM AND SYNCHRONIZATION

Sourcebank[realline] || [ realline ]| Targetbank[realline]

This instance of the Sourcebank process (with realline replacing each appearance of the formal gate ?) then could contain an offering of the form

realline ! tar-account-no ! amount

while the Targetbank (with realline in place of $S$) could contain

realline ? acc-no ? amnt

with the understanding that the appropriate account is to be credited with the received amount.

Now, consider augmenting the bank system with an auditor that records each transaction for record-keeping purposes. This can be modeled by a three-party synchronization, simply adding the auditor to the previous behavior:

(Sourcebank[realline] || [ realline ]| Targetbank[realline])
[[ realline]] Auditor[realline]

In this system an action involving realline may only occur if all three processes synchronize. Sourcebank presumably will offer values as before, while both Targetbank and Auditor are willing to accept such a pair of values.

Such multiparty interactions/synchronizations can be used to model a level of abstraction where a collection of processes come together, exchange information, and then continue, as one atomic action. This would presumably be implemented by an entire protocol to coordinate and exchange the necessary information.

In terms of the graphs of the individual processes, an interleaving parallelism means that at each point either of the processes can execute a step, and the graphs are thus merged. The parallelism that synchronizes at a gate means that the steps involving a gate in the interface are a step in both graphs, atomically.

As an additional notational convenience, $A || B$ denotes a partial synchronization where the union of all gates appearing in $A$ or $B$ are in the synchronization list between them. In other words, all visible
actions at gates must synchronize in A and B. This is known as synchronous parallelism.

The synchronization requirements may not always be satisfied by every behavior. In the extreme case of synchronous behavior, it is sufficient that the two processes can only continue with actions at different gates, and no internal action is possible. In that case, the system can deadlock, i.e., reach a state in which no action is possible. This is equivalent to a behavior with a stop action at the point of the deadlock. In fact it is possible to reason about whether a LOTOS behavior is equivalent to one with an explicit stop in order to determine whether deadlocks are possible.

Even when only partial synchronization is required, a deadlock may occur if each process can only continue with actions at gates that synchronize with parallel processes, but the parallel processes are not at the appropriate gate. Moreover, if the same synchronization gate has been reached in both processes, but the values offered and possible acceptances are not consistent as to their data type or possible values, the system can deadlock.

A simple deadlock situation is described by:

\[
(\text{coord} \! 0) \mid [\text{coord}, \text{othergate}] \mid (\text{othergate} \ ? x)
\]

Clearly, neither process can proceed, and the behavior is equivalent to stop. A more complex situation can be seen in

\[
(\langle i \ [] \text{coord} \! x \rangle ; i) \mid [\text{coord}] \mid (\text{coord} \ ? y)
\]

Here the behavior can either choose to do the synchronization action in both parallel components, where the left process then continues with internal actions, or choose to do the internal action of the left process, followed by other internal actions. In that case the right process cannot continue, since there the action on the coord gate must be synchronized. There is one trace that can properly terminate but the other will deadlock, since one parallel component cannot continue.

### 7.5 Additional composition of processes

A process is built from smaller behavioral expressions, composed using the semicolon and choice operators. Once processes are defined, more
complex behaviors can be described using (in effect) tail recursion, e.g., that one process continues to behave like another. Parallel compositions have been seen in the previous section.

Two additional forms of composition are used in LOTOS. First, as indicated in the description of the exit behavior, two processes can be sequentially composed. Note that this differs from the semicolon, which merely precedes a behavior by an action. The notation \( P \gg R \) means that after \( P \) properly terminates (by an exit behavior) then the continuation is the behavior of \( R \).

The behavior that connects the successful termination of \( P \) and the initialization of \( R \) is viewed as an internal action \( i \). In the graphic representation, the tree representing the sequential composition is obtained by replacing each exit transition of \( P \) (that always leads to a terminal node), by a transition labeled \( i \) leading to the tree representing \( R \). To treat the transfer of values implied by the functionality of \( P \), the full notation is

\[
P \gg \text{accept variable-list in } R
\]

Here variable-list is a list of variables and their types that will be used within \( R \), and must exactly match the functionality of \( P \). The sequential composition operator can be viewed as a special case of ‘parallel’ composition in which the first process synchronizes with the second one at a special gate at which the values in the accept list are transferred. The synchronization occurs when the first process reaches an exit statement, while the second is at its initialization point. This synchronization is hidden from the environment, since it is viewed as an internal action.

A second type of composition represents interruption of one behavior by another. This is denoted by \( P \triangleright R \). It means that the behavior of \( P \) may at any point continue with the behavior of \( R \). Graphically, the tree representing \( R \) is added as a nondeterministic choice at every point in the tree of \( P \) except at the leaves following an exit statement. When \( P \) successfully terminates, it is no longer possible to continue with \( R \). Moreover, once \( R \) is begun, the behavior does not return to that of \( P \).
7.6 A few words on Semantics

Although a full description of the semantic machinery associated with process algebras is beyond the scope of this book, the basic ideas can help shed light on the problems of concurrency. The semantic considerations in process algebras are often treated using simpler versions of the language that factor out all aspects connected to variables and data manipulation. In particular, there is a basic LOTOS notation in which only a gate name is listed, with no indication of values offered or accepted.

The intuitive approach seen in the graphic representation is to consider the collection of possible action traces as the semantics of a LOTOS process, resembling the semantics of state machines. This may be sufficient for some views of a system. In particular, when the only question is what computations are possible, this view, known as the trace semantics may be used. However, this is overly simplistic when deadlock situations are possible, or when one behavior is to be substituted for another. Now that there is both an explicit STOP action, as well as possible mis-matches on gates, it is not always enough to merely consider the successful computations.

In one semantic view, we seek to show that two behaviors are equivalent if each can simulate the other. That is, considering the graphs representing each behavior, a simulation relation $R$ is established between the nodes of the graphs. For nodes $a$ in graph $A$ and $b$ in graph $B$ that are related by $R$, if a sequence of actions $s$ can occur from state $a$ in $A$ and leads to $a'$, then $s$ leads from $b$ to $b'$ in $B$, and $a'$ and $b'$ must also be in the relation $R$. If there is a simulation relation in both directions, the corresponding behaviors are said to be bisimulation equivalent. In one variant, known as weak bisimulation, internal actions of each process can be ignored.

Another view is known as failure or testing equivalence. In this approach, the actions that can be refused by a behavior at various points are considered explicitly. Thus $B$ reduces $A$ is true if $B$ can only execute actions that $A$ can also execute at corresponding points and $B$ can only refuse to execute actions that $A$ can (potentially) also refuse. This type of refinement relation (or equivalence if the refinement holds in both directions) insists that there must be a sequence of (externally
visible) events (called a test) that can distinguish between two behaviors that are not equivalent. The test can include NOT responding to an action.

LOTOS does not require that one particular notion of equivalence be used. In fact, the different possibilities correspond to different requirements for implementability, and almost all of them could be reasonable, depending on the context.

Consider a process PH intended to represent a philosopher in a classic dining philosophers design. Such a process could have possible interaction gates called \( t \) (think), \( h \) (hungry), \( g \) (give-up), or \( e \) (eat). Three alternative definitions could be

\[
\begin{align*}
\text{PH}_1 &= t;h;g;\text{PH}_1 \parallel t;h;e;\text{PH}_1 \\
\text{PH}_2 &= t;(h;g;\text{PH}_2 \parallel h;e;\text{PH}_2) \\
\text{PH}_3 &= t;h;(g;\text{PH}_3 \parallel e;\text{PH}_3)
\end{align*}
\]

We then can ask whether the definitions above are equivalent or not. The answer depends on which semantic definition we prefer. If trace semantics are used, all three processes above are equivalent because they have the same traces. If a round consists of either \( t;h;g \) or \( t;h;e \), the traces of each definition above are an infinite iteration of rounds.

However, if a testing semantics is used, then \( \text{PH}_1 \) remains equivalent to \( \text{PH}_2 \), but both are not equivalent to \( \text{PH}_3 \). Consider a test that infinitely often repeats \( t;h;e \) in parallel with each one of the definitions. \( \text{PH}_3 \) will always succeed with this test: when the nondeterministic decision point is reached, only the right branch where the \( e \) actions match can be taken, and thus it must be chosen. However substituting either of the other processes can lead to either success or failure. There is a computation where the \( h \) of the test is matched with the left \( h \) of \( \text{PH}_2 \) and then progress cannot be made because the \( g \) action of \( \text{PH}_2 \) does not match the \( e \) action of the test for that execution. A similar “bad choice” can occur for the alternative \( t \) actions in \( \text{PH}_1 \).

There is no test that can differentiate between the two processes \( \text{PH}_1 \) and \( \text{PH}_2 \), because both succeed for the two steps \( t;h \), and then can either succeed or fail for the next step, depending on the nondeterministic choices made earlier.

Under a bisimulation semantics, all three processes above would be considered different. Any attempt to find matching pairs of nodes
so that the continuations have the same action sequences and reach matching points—will not succeed. For example the point after the $t$ in \textit{PH}2 does not correspond to any one point in \textit{PH}1, and the point after $t;h$ in \textit{PH}3 does not correspond to any point in either \textit{PH}2 or \textit{PH}1.

One way to incorporate such notions of equivalence is through the use of automatic tools that check whether two LOTOS specifications are equivalent, or related by a refinement relation. Another approach gives equivalences that are true for the semantic notion desired. For basic LOTOS, assuming $a$ and $b$ are different actions and $X$, $Y$, and $Z$ are processes, a few of the defining equivalences for interleaving and full synchronization are:

1. $X \ll STOP = X$
2. $X \parallel STOP = STOP$
3. $a;X \parallel b;Y = STOP$
4. $a;X \parallel a;Y = a;(X \parallel Y)$
5. $a;X \parallel b;Y = a;(X \parallel b;Y) \parallel b;(a;X \parallel Y)$
6. $(X \ll Y) \parallel Z = (X \parallel Z) \parallel (Y \ll Z)$

A proof that, for example, $a;b \parallel a;c = a;STOP$ then can be done easily by applying the equivalences 4 and then 3. These equivalences hold for all of the semantic definitions described above. Adding the equivalence

$$a;(X \ll Y) = a;X \ll a;Y$$

would be justified if a trace equivalence view is used, but would not be valid for a bisimulation equivalence semantics. (Why?)

### 7.7 Observations on style and use of LOTOS

There is a semi-mechanical translation from a state-machine to a LOTOS specification. In this translation, each state corresponds to a process, and the arrows (transitions) leaving that state correspond to alternative actions that can be taken. If the transition has side effects as in statecharts, those are listed as actions, and then the continuation is the process that corresponds to the target state. In this way,
the examples from the chapter on statecharts can almost be literally translated into LOTOS.

For example, a LOTOS process showing changes in the security levels of a system can be given that is almost a direct translation of a statechart or state machine version.

```plaintext
PROCESS
SECURITY : NOEXIT :=
LEVEL0
; ( key1 ; LEVEL1
   []
   key2 ; LEVEL2
 )
; ( reset [] timeout )
; SECURITY
```

The process `SECURITY` is defined as being nonterminating (using the keyword `NOEXIT`), and to be composed of the behavior of first being in `LEVEL0` followed either by synchronizing with the external gate `key1` or `key2`, and entering the appropriate level, which is then followed by either a synchronization with an external `reset` event or a timeout (here merely an uninterpreted event, that would have to be given the appropriate meaning in the other process containing that gate). That event is followed by a behavior that corresponds to that of the entire process again (i.e., starting with `LEVEL0`).

The parallel process that corresponds to taking a desired action only when `LEVEL2` has been reached can be described using a gate that communicates a security level of 2, to be communicated from the `LEVEL2` process.

Such a direct translation from a state machine or statechart is only one possible style for LOTOS, and does not fully take advantage of the potential of the language. In particular, it is extremely useful to exploit the abstract process definitions. For example, it is typical to have a process `REGISTER` in describing the architecture of a system. Such a process has an appropriate data type defined algebraically, but also has concurrent events and interactions with other parts of the system that affect it. The various registers and their connections could
then be seen as instantiations of the \texttt{REGISTER} process with different names for parameters and gates.

Moreover, hiding internal operations using the \texttt{i} action and not specifying all of the processes in advance can be exploited to provide multiple levels of specification. At the level of system architecture, it is possible to merely list the names of the components and the gates on which they synchronize. For example, in a system with dual processors, the collection of messages that must be treated might appear as:

\begin{verbatim}
front| [alert, standard, newreq,...] |back
\end{verbatim}

Then subsequent refinements could gradually add specifications for the internal workings of the processors, leaving some aspects as internal i actions.

The LOTOS language has been adopted as an ISO standard for protocol and software specification. This follows a significant effort on the part of the developers of the language in this direction, and does not necessarily mean that the language is preferable to others. The developers felt that the best way to introduce a formal notation was first to reach agreement on a standard, introduce it into accepted product development and documentation, and then follow with tools.

Most experience with the language is in Europe, and there it is considered precise and useful. There have been criticisms that the language is difficult to learn, and, at least as commonly used, rather low level. In addition to the academic development projects described below, LOTOS is used in specifications of telephone switches, and is being considered within the European Esprit research initiative. A WWW home page on many formal methods, including pointers to LOTOS, is available at

\begin{verbatim}
http://www.comlab.ox.ac.uk/archive/formal-methods.html
\end{verbatim}

and updates on the status of analysis and simulation tools may be found there.

LOTOS has been slow to develop automatic aids and tools, but today there is considerable effort in this direction. In particular, the developers have followed an opposite course to that seen in some less formal graphic notations. In those languages, first a graphic notation
existed, and only later were attempts made to formalize the notation to enable analysis tools. In LOTOS, the foundations are mathematically precise, and a textual notation is fully developed. Only recently have graphic versions (one of which is called gLOTOS) been developed in order to increase the accessibility of the language in industrial contexts. A primary effort in this direction is underway at the University of Madrid, as part of their Midas project, and at the University of Twente in Holland, where the MiniLite project has similar goals. At the University of Grenoble LOTOS has been integrated into their analysis tools for distributed systems.

The LOTOS notation can be abstract, but the commitment to processes and gates already at early stages can be arbitrary. For this reason the notation is sometimes considered appropriate for design rather than more abstract requirements specifications.

### 7.8 Bibliographic remarks

Process algebras were first suggested by Robin Milner [6] in the CCS language, and by C.A.R. Hoare in CSP [3]. Both of these have been used to investigate the nature of concurrency, and how it should best be modeled in a formal notation that encourages compositionality. Both have been used to specify systems, particularly CCS.

Using the concepts introduced in CCS and CSP, LOTOS was introduced by Bolognesi and Brinksma in [1], specifically for ‘industrial strength’ specification and analysis. For example, some of the parametric gates, and the continuum of synchronization levels are loosely based on similar constructs in the other languages, but seem easier to form into libraries due to the naming conventions. An introductory tutorial can be found in [5]. Considerable effort has been devoted to making LOTOS an ISO standard for communication protocols, including the ACT-ONE algebraic specification language as a medium for defining data relations within LOTOS.

Tools and methodology for using LOTOS can be found in [2]. Additional tools for the analysis of process algebra specifications, and for showing equivalence of process algebra expressions, for a variety of equivalence definitions, are described in [4].
Bibliography


