The Formal Specification Language

LOTOS

A Course for Users

Kenneth J. Turner

Department of Computing Science
University of Stirling
The Formal Specification Language

LOTOS

A Course for Users

© Kenneth J. Turner

16th April 1996

Department of Computing Science
University of Stirling
Stirling FK9 4LA
Scotland

Telephone: +44-786-73171 Facsimile: +44-786-63000 Telex: 777557 STUNIV G
Eurokom: Ken Turner UST Email: kjt@cs.stir.ac.uk

This material is the basis of a forthcoming book to be published by MacMillan. Please do not copy it further.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.8.3</td>
<td>Events</td>
<td>22</td>
</tr>
<tr>
<td>2.8.4</td>
<td>LOTOS-Speak</td>
<td>23</td>
</tr>
<tr>
<td>2.9</td>
<td>Possible Solutions</td>
<td>24</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Synchronisation Tree</td>
<td>24</td>
</tr>
<tr>
<td>2.9.2</td>
<td>One-Shot Dictionary</td>
<td>24</td>
</tr>
<tr>
<td>2.9.3</td>
<td>Events</td>
<td>25</td>
</tr>
<tr>
<td>2.9.4</td>
<td>LOTOS-Speak</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>Basic Value Expressions</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Values</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>Local Definitions</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Local Variables</td>
<td>28</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Local Processes</td>
<td>28</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Domains</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Guards</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Events with Values</td>
<td>31</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Action Denotation</td>
<td>31</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Synchronisation Rules</td>
<td>33</td>
</tr>
<tr>
<td>3.5</td>
<td>Processes with Values</td>
<td>34</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Parameters and Results</td>
<td>34</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Functionality</td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Summary</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>Exercises</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>Possible Solutions</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>Data Typing</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Abstract Data Types</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Type Definitions</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Operations</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Equations</td>
<td>44</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Basic Equations</td>
<td>44</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Conditional Equations</td>
<td>45</td>
</tr>
<tr>
<td>4.5</td>
<td>Overloading</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>Making New Types</td>
<td>48</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Renaming</td>
<td>48</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Parameterisation</td>
<td>48</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Actualisation</td>
<td>49</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary</td>
<td>50</td>
</tr>
<tr>
<td>4.8</td>
<td>Exercises</td>
<td>51</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Improper Fractions</td>
<td>51</td>
</tr>
<tr>
<td>4.8.2</td>
<td>Washing Machine</td>
<td>51</td>
</tr>
<tr>
<td>4.8.3</td>
<td>Set of Incrementable Bits</td>
<td>52</td>
</tr>
<tr>
<td>4.9</td>
<td>Possible Solutions</td>
<td>52</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Improper Fractions</td>
<td>52</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Washing Machine</td>
<td>53</td>
</tr>
<tr>
<td>4.9.3</td>
<td>Set of Incrementable Bits</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>Putting It Together</td>
<td>57</td>
</tr>
<tr>
<td>5.1</td>
<td>Complete Specifications</td>
<td>57</td>
</tr>
<tr>
<td>5.2</td>
<td>Semantic Basis</td>
<td>58</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Process Semantics</td>
<td>58</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Data Type Semantics</td>
<td>59</td>
</tr>
<tr>
<td>5.3</td>
<td>Summary</td>
<td>60</td>
</tr>
<tr>
<td>5.4</td>
<td>A Large Example</td>
<td>61</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Network Service</td>
<td>61</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Data Transfer</td>
<td>61</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Interaction Point ................................................................. 4  
1.2 Synchronisation Tree ............................................................ 5  
1.3 Example of a Relation ........................................................... 7  

2.1 Synchronisation Tree with Initial Choice ................................. 16  
2.2 Synchronisation Tree with Deferred Choice ............................. 17  
2.3 Possible Synchronisation Tree ................................................. 24  

5.1 Network Service Access Points .............................................. 61  
5.2 Network Connection End-Points ............................................. 61  
5.3 Network Service Primitive Types .......................................... 62  
5.4 Network Service Constraints ................................................. 64
LIST OF FIGURES
Preface

LOTOS is, of course, the product of a protracted standardisation process involving many people internationally. LOTOS has been defined by Sub-Group C of the joint ISO+IEC Sub-Committee JTC1/SC21/WG1, under the Chairmanship of Ed Brinksma (University of Twente, Netherlands).

The production of an early version of these lecture notes was supported by the CEC (through the Esprit SEDOS project), and by ICL (through the QUASARS project). I am grateful to Alastair Tocher (International Computers, UK) for his comments on the early version, and for the many students who have helped to debug them.
Chapter 1

Introduction

1.1 Scope and Objectives

- this course is intended to introduce all features of LOTOS
- the course conforms to the definition of LOTOS given in the International Standard (published in 1989)
- the course is intended for users of LOTOS, mainly those who wish to read and understand LOTOS specifications; the course therefore concentrates on developing an intuitive grasp of the language by presenting basic concepts, backed up by suggestive examples
- to the uninitiated, formal languages (and LOTOS is no exception) appear abstruse and impenetrable; it is hoped that readers who complete this course will have overcome their ‘culture shock’, and will feel able to pick up specifications in LOTOS and get something out of them
- it should be recognised, however, that LOTOS is a formal language; an intuitive understanding of the language is therefore not a substitute for understanding the implications of the formal semantics which underlie it
- this course is therefore only an introduction to LOTOS as defined in the ISO standard
- throughout the course, copious examples have been used; these have intentionally been kept light (and, at times, whimsical) in an attempt to keep the interest of the reader; it would be wrong to conclude that LOTOS was applicable to toy problems only
- the reader of the course is encouraged to try the examples; equally, the reader is encouraged to manufacture other examples which are relevant to day-to-day work; the reader should avoid trying to specify anything too complex at first

1.2 Background and Context

1.2.1 OSI

- ISO (International Standardisation Organisation) is producing Standards for OSI (Open Systems Interconnection) in response to the demands of manufacturers and users for compatible interconnection of data processing equipment
- OSI is a major 10-year effort, with thousands of man-years in design and specification to produce a rich variety of telecommunications standards for interconnection of heterogeneous equipment
- for the goal of compatible interconnection to be achieved, precise and clear specifications are needed; however, the vast majority of ISO specifications are currently in natural language
- there are major problems in achieving compatible interconnection:
– how can ISO ensure that thousands of implementers world-wide will interpret OSI standards in a compatible way?
– who will provide the definitive interpretation when the original developers of the standards have dispersed?

1.2.2 Formal Description Techniques

. the scale and complexity OSI led to the formation of a group to standardise formal specification languages for OSI
. ISO developed FDTs (Formal Description Techniques) to provide the basis for unambiguous interpretation of standards
. FDTs are required to help:
  – specifiers — precision, conciseness, clarity, verifiability
  – implementers — avoidance of over-specification, clear guidance as to what, not how
  – testers — isolation of implementation options, basis of rigorous testing against specifications
. work in ISO committee SC21/WG1 (OSI Architecture) was established in order to develop FDTs for OSI
. initially, around 20 different techniques were proposed, in two broad categories:
  – finite state-machine techniques, which led to ESTELLE (Extended Finite State-Machine Language)
  – algebraic techniques, which led to LOTOS (Language Of Temporal Ordering Specification)
. three FDT Sub-Groups were set up:
  – A - Architectural aspects of FDTs
  – B - ESTELLE
  – C - LOTOS
. a later collaboration on FDTs was set up with SGX/WP3 of CCITT (International Telephone and Telegraph Consultative Committee)
. CCITT have also standardised SDL (Specification and Description Language)
. the net result is that there are three officially-recognised FDTs in ISO and CCITT

1.2.3 Development of LOTOS

. Sub-group C initially adopted CCS (Calculus of Communicating Systems)
. some notation was later introduced from CSP (Communicating Sequential Processes)
. the development of LOTOS was based on architectural principles
. a prototype of the language was used on sample OSI Standards, leading to improvements in the language
. the ACT ONE data typing language was added to allow formal and abstract specification of data types
. LOTOS is the first piece of mathematics to be standardised internationally
. an International Standard (8807) for LOTOS was completed in August 1988, after 8 years’ work!
1.2.4 **The Nature of LOTOS**

- LOTOS is a specification language, with a formal basis
- ESTELLE and SDL are semi-formal implementation languages, useful for describing reference implementations of OSI standards
- LOTOS is widely supported by academic institutions world-wide, and has significant industrial support
- LOTOS was designed for the specification of OSI systems, but is equally suitable for the specification of concurrent or distributed systems generally
- the formal basis of LOTOS ensures precision and analysability; however, this is bought at the price of needing special training in the language, and more brain-work to understand LOTOS specifications
- using LOTOS demands time and attention:
  - specification languages have a different philosophy from programming languages and natural languages
  - expressing oneself precisely requires discipline
  - thinking about what one really means is hard work

1.2.5 **Application of LOTOS**

- LOTOS has been used mainly on OSI, but this is not an intrinsic limitation
- LOTOS is applicable to sequential, concurrent, and distributed systems generally
- the following LOTOS specifications of OSI have been written (CL = Connection-Less, CO = Connection-Oriented)
  - **Application Layer**: FTAM (File Transfer and Manipulation) and ACSE (Association Control Service Elements)
  - **Presentation Layer**: some work (CO)
  - **Session Layer**: complete (CO)
  - **Transport Layer**: complete (CO)
  - **Network Layer**: Service (CL, CO) and Protocol (CL)
  - **Data Link Layer**: Service (CL, CO)
- LOTOS is being used on new OSI developments (e.g. ODP (Open Distributed Processing) and OSI Management)

1.2.6 **Related Formal Languages**

- LOTOS is an algebraic specification language, which includes abstract data type languages such as ACT ONE, and process algebras such as CSP and CCS
- notable features of LOTOS are:
  - its integrated data typing and behavioural modelling
  - its handling of non-determinism and concurrency
  - its orientation towards OSI
- LOTOS data typing is similar to other languages such as CLEAR and OBJ
- LOTOS uses CSP-like notation for events, ‘!’ and ‘?’, but these are more than just ‘input’ and ‘output’
- this gives LOTOS powerful synchronisation features not in CCS or CSP (e.g. multi-way synchronisation or value negotiation)
LOTOS gates correspond to CSP channels; LOTOS processes interact synchronously, just as in CSP and CCS.

In LOTOS, unlike in CCS and CSP, it is gates not whole events which are hidden, passed as parameters, etc.

LOTOS has a ‘kind choice’ operation (CSP ‘[ ]’ or CCS ‘+’)

LOTOS does not have the CSP non-deterministic choice operation ‘.’

For non-determinism, LOTOS follows the CCS approach of an internal event (CCS ‘ø’)

LOTOS has enabling and disabling operators in addition to basic CCS or CSP ones

LOTOS has CSP parallel composition, which forces synchronisation: there is no need to restrict and hide events for synchronisation, as in CCS.

1.3 Basics of Process Algebra

1.3.1 Processes

A process is a component of a specification; it is the abstraction of an activity in an implementation, and communicates with other processes.

A process is considered to be a black-box at some level of abstraction; only the external behaviour of a process is considered.

Processes share a communication mechanism called an interaction point, as shown in Figure 1.1; this is the abstraction of an interface in an implementation.

![Interaction Point](image)

In LOTOS, the specification concept of interaction point corresponds to the language concept of event gate (or just gate).

Processes are specified by giving a behaviour expression which defines their externally visible behaviour in terms of the permissible sequences of events in which they may participate.

In these course notes, processes are given upper-case names such as:

ACTIVATE_ALARM    DATA_TRANSFER

1.3.2 Events

An event represents a synchronisation between processes.

An event offer represents the ability of one process to participate in an event; events external to a process are resolved in conjunction with the environment of the process.

An event normally requires the participation of two or more processes; three kinds of event are possible:
1.3. BASICS OF PROCESS ALGEBRA

- pure synchronisation - no values are exchanged between the processes
- value establishment - one or more processes supply a specific value which lies in the set acceptable to the other process(es)
- value negotiation - two or more processes agree on a set of values

• events are considered to be atomic, i.e. at the level of abstraction of the specification, the synchronisation of the processes and the associated information are established at the same time; in an implementation, an event may of course correspond to a sequence of elementary steps

• the set of events in which a process can potentially participate is called the alphabet of the process

• the set of events in which a process can immediately participate at any point in the unfolding of its behaviour is called the initials of the process

• in the earlier parts of these notes, events with only an event gate are considered; more complex structures for events are considered later

• a behaviour expression is evaluated in an environment which offers it events to synchronise on; the environment may be another behaviour expression which it is combined with, or may be something external

• if synchronisation on event offers from the environment is not possible, deadlock has occurred (cf. a deadly embrace or a terminated process)

• if no events are offered by a behaviour expression, livelock has occurred (cf. looping)

• in these course notes, events are given lower-case names such as:

  \texttt{timer\_expired} \quad \texttt{ready}

1.3.3 Temporal Ordering

• specifications in LOTOS give the temporal ordering of events, i.e. the relative ordering of events in time

• LOTOS abstracts away from absolute timing considerations, e.g. that an event must occur at a specific time or after a specific period

• LOTOS specifications use operators which combine behaviour expressions to yield more complex behaviour expressions; these operators obey well-defined laws which enable any specification to be interpreted unambiguously

• the data typing part of LOTOS is given semantics using the theory of algebras; LOTOS operators for combining behaviour expressions can be shown to satisfy algebraic laws which characterise them

• a convenient diagrammatic notation for thinking about behaviour in languages like LOTOS is the synchronisation tree, as shown in Figure 1.2

  \begin{figure}[h]
  \centering
  \begin{tikzpicture}
    \node (e) {\texttt{e}}; \\
    \node[below of=e] (e1) {\texttt{e}_1}; \\
    \node[below of=e] (e2) {\texttt{e}_2}; \\
    \node[below left of=e] (choice) {\texttt{choice between events \texttt{e}_1 or \texttt{e}_2}}; \\
    \node[below right of=e] (occurrence) {\texttt{occurrence of an event \texttt{e}}}; \\
  \end{tikzpicture}
  \caption{Synchronisation Tree}
  \end{figure}


1.4 Background Needed

1.4.1 Differences from Programming

• surprisingly little technical knowledge is needed to achieve some familiarity with LOTOS; a numerate back-
ground, with a small amount of basic mathematics and computing, is probably sufficient

• a knowledge of programming languages will also help, but beware that LOTOS being a specification language,
treats some concepts in a fundamentally different way

• variables in LOTOS are mathematical variables: they are simply names which happen to be bound to a
particular value in particular, one does not assign a value to a LOTOS variable as one might assign a value to
a store location; in LOTOS one cannot write, for example:

\[ x := x + 1 \]

one must in effect define a new variable (\( x' \), say) which is bound to the value of the old \( x \) plus 1

• recursion in programming languages is usually thought of in terms of a stack which records return addresses;
a well-behaved program unwinds this stack before continuing

• in LOTOS, however, one may exit from within a recursive piece of behaviour to do something completely
different; this is because recursion in LOTOS is equivalent to replacing a recursive call with a copy of the
text defining the behaviour

• efficiency is an important consideration in programming, but is not appropriate in LOTOS; one should not
think of ‘implementing’ LOTOS literally, and must therefore not be concerned about matters of efficiency

• a good specification clearly defines what and not how; this contrasts sharply with programming, where one
frequently makes design decisions to optimise the use of store, processor, etc.

• over-specification must be carefully avoided in LOTOS; one must always concentrate on external requirements
and black-box behaviour, not structures or algorithms to implement these

• for example, one should specify a queue in terms of the operations on it (enqueue, dequeue, head, etc.)
rather than in terms of linked-lists, algorithms for skipping down the queue, etc.

1.4.2 Sets and Relations

• a set can be thought of as an unordered collection of elements in which duplicates are ignored (this glosses
over certain mathematical niceties)
sets are usually written in curly brackets, for example:

\{Kenneth; John; Turner\}

• the empty set, {}, has no elements

• each element in a set is a member of that set; one set is included in another if its members are also members
of the other set; the first such set is a subset of the second

• two sets are equal if they have the same members (i.e. each is a subset of the other)

• the cartesian product of a number of sets is the set of all tuples (ordered lists) drawn from the sets; the
symbol \( \times \) is used for this ‘multiplication’ operation, for example:

\( \{run; fall\} \times \{down; off; out\} \)
is:

\( \{< run; down >; < run; off >; < run; out >; < fall; down >; < fall; off >; < fall; out >\} \)
1.4. BACKGROUND NEEDED

- the powerset, $\mathcal{P}$, of a set is the set of all its subsets; for example:

$$\mathcal{P}\{p; i; g\}$$

is:

$$\{\emptyset; \{p\}; \{i\}; \{g\}; \{p; i\}; \{p; g\}; \{i; g\}; \{p; i; g\}\}$$

- a relation between two sets is a set of ordered pairs of elements from the two sets (it is a subset of their cartesian product); for example, Figure 1.3 corresponds to:

  $\langle$ lemon; yellow $\rangle$; $\langle$ lime; yellow $\rangle$; $\langle$ lime; green $\rangle$; $\langle$ apple; green $\rangle$;
  $\langle$ apple; red $\rangle$; $\langle$ longan; stramineous $\rangle$

![Figure 1.3: Example of a Relation](image)

1.4.3 Functions

- a function is a relation which maps an element in one set to exactly one element in another set (several elements may get mapped onto the same element)

- the set to which the function is applied is its domain; the set to which the function maps is its codomain; the following mathematical notation is used for this:

$$F : D \rightarrow C$$

where $F$ is the function, $D$ is its domain, and $C$ is its codomain

- a function may also map from several sets to several sets; for example, in integer arithmetic:

  $$\text{divide} : \text{dividend} \times \text{divisor} \rightarrow \text{quotient} \times \text{remainder}$$

  which has as ‘arguments’ the number to be divided and the divisor itself, and has as ‘results’ the quotient and remainder

- however, functions in LOTOS are allowed to have only one named set as codomain; one solution to this is to define separate functions for each codomain; for example:

  $$\text{divquot} : \text{dividend} \times \text{divisor} \rightarrow \text{quotient}$$
  $$\text{divrem} : \text{dividend} \times \text{divisor} \rightarrow \text{remainder}$$

  such functions are called operations in LOTOS
• a \textit{total function} or \textit{operation} gives a mapping for each element in its domain, otherwise it is \textit{partial}

• an \textit{associative} operation is one in which the bracketing is irrelevant; for example, in ordinary arithmetic:

\[(a \cdot b) \cdot c = a \cdot (b \cdot c)\]

and so may be written without ambiguity as:

\[a \cdot b \cdot c\]

• a \textit{commutative} operation is one in which the order of the operands does not matter; for example, in ordinary arithmetic:

\[a \cdot b = b \cdot a\]

1.4.4 Syntax and Semantics

• syntax is usually expressed using a \textit{grammar}; this consists of a set of rules in which \textit{non-terminal symbols} (variables of the grammar) are defined in terms of non-terminal symbols and \textit{terminal symbols} (constants of the grammar)

• the definition of LOTOS syntax includes the following constructs:
  
  – recursion is expressed by, for example:
    
    \texttt{echo = "hello" echo}

    which defines \texttt{echo} to be \texttt{hello} (a terminal symbol) followed by \texttt{echo}; in other words, \texttt{echo} is an unbounded repetition of \texttt{hello}

  – a choice is expressed by, for example:
    
    \texttt{mood = happy \mid sad}

  – an optional part of a rule is expressed by, for example:
    
    \texttt{sentence = subject verb \{object\}}

    which says that zero or one occurrences of \texttt{object} are allowed

• a set of \textit{derivation rules} is a system of logic for dealing with inferences; such a system is used to express the semantics of LOTOS operators

• a derivation system has a set of \textit{axioms} (logical formulae from which others are derived), and a set of \textit{assertions} (logical formulae which may or may not be derivable)

• a logical formula is derived by applying \textit{inference rules} of the form:

\[
P_1; \ldots; P_n \quad \Rightarrow \quad Q
\]

meaning that, given \(P_1\) up to \(P_n\), \(Q\) may be derived

• the shorthand notation:

\[
\vdash P
\]

means that \(P\) can be derived from the axioms and the inference rules
1.5 Summary

- LOTOS is a formal specification language being developed by ISO for the specification of OSI Standards.
- LOTOS is generally applicable to the specification of concurrent and distributed systems generally.
- LOTOS is an algebraic specification language which describes systems by giving their externally visible behaviour in terms of the permissible sequences of events they may participate in.
- LOTOS specifications consist of behaviour expressions (often generalised as processes) which synchronise on events at gates.
- Events may be pure synchronisation, value establishment, or value negotiation.
- The alphabet of a behaviour expression is the set of all its potential events, whereas its initials are those it can immediately offer.
Chapter 2

Basic Process Expressions

2.1 Sequence

- the *sequential composition operator* `;` is used to prefix a behaviour expression with an event called an *action prefix*; for example:

  button_pressed; RING_BELL

- action prefixes associate to the right; for example:

  connect_request; connect_confirm; data; DISCONNECT

  means:

  connect_request; (connect_confirm; (data; DISCONNECT) )

- the alphabet of this behaviour expression is:

  \{connect_request; connect_confirm; data\}

  plus whatever events there are in DISCONNECT

2.2 Choice

- the *choice operator* `[ ]` is used when alternative behaviours are allowed; for example:

  (lift_arrived; ENTER) [ ] (lift_broken; USE_STAIRS)

- choice is associative (as one would expect); for example:

  EAT [ ] DRINK [ ] BE_MERRY
means the same as:

(EAT [] DRINK) [] BE_MERRY

and:

EAT [] (DRINK [] BE_MERRY)

but notice that one cannot eat, drink, and be merry!

• choice is also commutative (as one would again expect); for example:

PAY_A_FINE [] TAKE_A_CHANCE

means the same as:

TAKE_A_CHANCE [] PAY_A_FINE

• the initials of the behaviour expression:

(lunch_bell; EAT) [] (boss_here; WORK) [] (fire; PANIC)

are:

{lunch_bell; boss_here; fire}

• in the straightforward case, the choice between alternatives is resolved by the environment of the process; in the example above, if the environment offers only fire then the process will PANIC

• more complicated cases involving non-determinism are considered later in these notes

2.3 Parallelism

2.3.1 Interleaving

• the interleaving parallel composition operator ‘||’ is used to allow behaviours to unfold completely independently in parallel; the events from each behaviour expression are interleaved for example:

(data_in; data_out; BUFFER) || (read; mark; digest; BOOK)

includes the following behaviours:

\[ data\_in; read; mark; data\_out; digest::: \]

\[ read; mark; digest; data\_in::: \]

• ‘||’ is associative and commutative (as one would expect, in order to capture the intuitive concept of running in parallel)
2.3.2 Synchronisation

- the synchronising parallel composition operator ‘||’ is used where there are events that need to be synchronised; in this case the events (strictly, gates) which occur in either of the behaviour expressions are obliged to synchronise
  
  for example:

  
  (bang; start; finish; ATHLETE)
  
  ||

  (bang; start; finish; STARTER)

  may engage in the following sequence of events:

  
  bang; start; finish; :::

- if only certain events are to be synchronised, the ‘[::]’ form of the operator is used, with the events (strictly, gates) named between ‘[’ and ‘]’; for example:

  
  (off_hook; dial; answer; speak; on_hook; TELEPHONE)
  
  [dial]

  (find_number; dial; engage_brain; speak; CALL)

  will synchronise only on the dial event, and will allow speak in the second behaviour expression before answer and after on_hook in the first behaviour expression (a possibly realistic situation)

- ‘[::]’ with an explicit empty list, or with no gates which occur in the two behaviour expressions, is equivalent to ‘||’

- ‘[::]’ with an explicit list which is in fact the union of the two sets of gates is equivalent to ‘||’

- like ‘||’, ‘|’ or ‘[::]’ is associative and commutative

- parallel composition can be used to express independent constraints; for example, the constraints ‘breakfast must precede lunch’ and ‘lunch must precede dinner’ can be expressed by:

  
  (breakfast; lunch; AM) [lunch] (lunch; dinner; PM)

  this allows a separation of concerns, not to mention eating habits!

2.4 Termination

2.4.1 Inaction and Success

- the simplest behaviour expression is ‘stop’, which offers no events and therefore does nothing; it is used to represent inaction or deadlock

- successful termination of a behaviour expression is represented by ‘exit’, which offers a special event ¥ and then behaves as stop

- ¥ is part of the underlying mathematical model of LOTOS and is not an event which can be explicitly offered; ¥ may also be an implied initial of a behaviour expression
• successful termination of a sequence (using the ‘;’ operator) depends on whether the right-most behaviour expression is stop or exit
  for example:

  clock_in; clock_out; exit

  may terminate successfully, but:

  born; died; stop

  can not

• successful termination of a choice (using the ‘[]’ operator) depends on the successful termination of one of the behaviour expressions; for example:

  (fail; stop) [] (catastrophe; stop)

  can not terminate successfully, but:

  (lift_off; exit) [] (armageddon; stop)

  may succeed

• successful termination of a parallel composition (using the ‘||’, ‘||’, or ‘:::’ operators) depends on the successful termination of all of the behaviour expressions; for all these operators, the special termination event ¡ is always synchronised — even in the case of ‘||’
  for example:

  (finished; exit) || (done; exit)

  may terminate successfully

2.4.2 Enabling

• as a generalisation of the ‘;’ operator, which is used for the sequential composition of an event and a behaviour expression, the ‘>>’ operator (pronounced enables) combines two behaviour expressions in sequence
  for example, if process SHOP is:

  visit_shop; buy_food; come_home; exit

  and process EAT is:
cook_food; eat_food; stop

then the process DINE, defined by:

SHOP >> EAT

denotes the possible sequence of events:

visit_shop; buy_food; come_home; cook_food; eat_food

• exit (i.e. the ; event) is absorbed by a following ‘>>’

• if the left-hand behaviour expression does not terminate successfully, the right-hand behaviour expression will not apply; for example:

(prime_5; prime_7; (prime_9; stop []) exit) >> (prime_11; exit)

can successfully terminate after the sequence of events:

prime_5; prime_7; prime_11

but will deadlock after the sequence of events:

prime_5; prime_7; prime_9

• ‘>>’ is associative, but is not of course commutative

2.4.3 Disabling

• a frequent occurrence in specifications is the need to specify behaviour which may be interrupted by something else (e.g. disconnection may terminate data transfer)

• the ‘[>’ operator (pronounced disabled by) allows the right-hand behaviour expression to interrupt the left-hand behaviour expression; if this happens, the future behaviour is that of the right-hand behaviour expression only

• if the left-hand behaviour expression terminates successfully, then the combination also terminates successfully (i.e. disabling is no longer possible); this is to allow for contingencies for example:

(send_data; reset_timer; receive_acknowledgement; exit) [>
(timer_expired; sound_alarm; stop)

may terminate successfully if an acknowledgement is received to a message, but may sound an alarm if no acknowledgement is received within some time period
• beware that behaviour may be disabled between its ‘last’ event and stop or exit
  thus, in the example above, timer_expired may occur before reset_timer and after receive_acknowledgement;
  this could be realistic in an actual implementation

• the left-hand behaviour expression is often non-terminating (iterative or recursive), and the right-hand one
  is often a closing-down behaviour expression; for example:

  DATA_TRANSFER [ > DISCONNECTION

• ‘[>’ is associative, but is not of course commutative

2.5 Non-Determinism

2.5.1 Non-Determinism due to Choice

• the events in a process specification are event offers; the actual events which happen may be influenced by
  the environment of the process
  for example:

  (wake_up; MAKE_COFFEE) [] (open_wine; GET_DRUNK)

  will result in MAKE_COFFEE or GET_DRUNK depending on the sobriety of the environment

• however, identical events may be offered as alternatives: in this case, the environment cannot influence
  which branch is taken; the choice is made non-deterministically
  for example:

  (eat_out; CHINESE_MEAL) [] (eat_out; INDIAN_MEAL)

  will result in CHINESE_MEAL or INDIAN_MEAL after the environment has offered eat_out, and cannot be
  influenced

• non-determinism is best understood by considering the synchronisation tree for the behaviour; for example:

  (a; b; stop) [] (a; c; stop)

  has the synchronisation tree shown in Figure 2.1

  \[
  \begin{array}{c}
  a \ \ast \ a \\
  b \ \ast \ \ast \ c
  \end{array}
  \]

  Figure 2.1: Synchronisation Tree with Initial Choice

but:
2.5. NON-DETERMINISM

\[ a; (b; \text{stop}) \] [c; \text{stop}]

has the synchronisation tree shown in Figure 2.2

\[ \text{Figure 2.2: Synchronisation Tree with Deferred Choice} \]

- in the first case, the environment offers \( a \) but has no choice as to what follows: it may be offered \( b \) or \( c \); in the second case, the environment can still decide on \( b \) or \( c \) after offering \( a \)

2.5.2 Hiding

- LOTOS supports top-down decomposition of behaviour: processes (specification components) can be progressively decomposed into simpler processes
- however, it is important that the details of this decomposition (in particular, any internal events) are not visible at a higher level
- the hiding operator ‘hide ... in’ is therefore used to hide events (strictly, gates) which are internal to the behaviour of a system; for example:

\[(\text{begin}; \text{middle}; \text{exit}) \] [\text{middle}] [\text{middle}; \text{end}; \text{exit}]\]

may engage in the sequence of events:

\[\text{begin}; \text{middle}; \text{end}; i\]

but:

\[\text{hide middle in} \ (b; \text{middle}; \text{exit}) \] [\text{middle}] [\text{middle}; \text{end}; \text{exit})\]

may engage in only:

\[\text{begin}; \text{end}; i\]

2.5.3 Internal Events

- a hidden event is represented by \( i \), the internal event; \( i \) may be an initial of a behaviour expression
  for example:

\[\text{hide second in} \ (\text{first}; \text{second}; \text{third}; \text{stop})\]
may engage in:

\[ \text{first; } \varepsilon \text{ third} \]

- the internal event represents non-determinism since the environment may not influence it; because the internal event may affect subsequent behaviour due to an internal choice it is useful to represent it because it ‘explains’ why a particular behaviour happened

\[ \text{WORK } [ ] (i; \text{GO\_TO\_BED}) \]

indicates that \text{GO\_TO\_BED} may happen without \text{WORK} being an option; the \text{i} represents a decision which cannot be influenced by the environment

this might arise from:

\[ \text{hide} \text{dawn\_chorus in (WORK } [ ] (\text{dawn\_chorus; GO\_TO\_BED}) \) \]

- once the internal event has occurred, only the behaviour following it is allowed
- it is quite legitimate for all branches to be ‘protected’ with an \text{i}; for example:

\[ (i; \text{WORK}) [ ] (i; \text{STRIKE}) [ ] (i; \text{GO\_ON\_HOLIDAY}) \]

- the internal event is one way of expressing implementation freedom (where there are different ways to do something), or some internal decision (which it is not appropriate to spell out at a given level of abstraction)
- internal events are indistinguishable from each other
- when \text{exit} combines with ‘\text{>}>’, the result is that the \text{i} event turns into an internal event

2.5.4 Behaviour Equivalence

- the introduction of the internal event raises questions of when two behaviour expressions mean the same thing
- two behaviour expressions are isomorphic if they can be transformed into each other by systematic renaming; for example:

\[ (\text{slithy; slithy; mimsy; BOROGOVE}) [ ] (\text{mimsy; slithy; BOROGOVE}) \]

is isomorphic to:

\[ (\text{lubricilleux; lubricilleux; emmime; GOUGEBOQUET}) [ ] (\text{emmime; lubricilleux; GOUGEBOQUET}) \]
two behaviour expressions are *observationally equivalent* if they exhibit the same behaviour ignoring the internal events; under this equivalence, any *finite* sequence of internal events can be ignored except in the context of a choice

for example:

\[
\text{aye; } i; i; \text{EYE}
\]

is observationally equivalent to:

\[
\text{aye; EYE}
\]

for example:

\[
(a; i; b; \text{exit}) > > ((c; \text{stop}) []) (d; i; \text{exit})
\]

is observationally equivalent to:

\[
a; b; ((c; \text{stop}) []) (d; \text{exit})
\]

• an infinite sequence of internal events is called *infinite chatter* and corresponds to *livelock*; for example, a process \text{LIVELOCK} may be defined as:

\[
\text{hide beep in (beep; LIVELOCK)}
\]

• *observational congruence* holds when one behaviour expression can be substituted for another in *all* contexts; for example:

\[
i; \text{PROC1}
\]

is not observationally congruent to:

\[
\text{PROC1}
\]

because it changes the overall meaning when substituted for it in a context such as:

\[
\text{PROC1} [] \text{PROC2}
\]
2.6 Behaviour Definitions

2.6.1 Definition and Instantiation

- A process is given a name and parameters by a behaviour definition; for example:

```
process GIVE_LECTURE
    [write_notes, give_lecture, understand_subject] : exit :=
    write_notes; give_lecture; understand_subject; exit
endproc
```

- A process may optionally be followed by a list of events (strictly, gates) inside ‘[...]’

- The keyword `exit` is used in the heading line if the process may terminate successfully; `noexit` is used if it can never terminate.

- The behaviour definition gives formal parameters to the process; the process is referred to in a process instantiation by giving actual parameters such as:

```
GIVE_LECTURE
    [plagiarise_notes_by_Ed, lecture_at_Stirling, understand_LOTOS]
```

- A process may also be optionally defined to have parameters and results; for example:

```
process PROC [gates] (parameters) : exit (results) :=
    ...
endproc
```

This is covered in detail later in these course notes.

2.6.2 Recursion

- A process may refer to itself in its definition; for example:

```
process PRESS_UP [up, down] : noexit :=
    up; down; PRESS_UP [up, down]
endproc
```

- It is important not to think of this kind of recursion in a programming language sense: there is no ‘subroutine stack’, which has to be unwound neatly; recursion is equivalent to repeated textual substitution of the definition of a process.
2.7 Summary

- (tail) recursion is the way of expressing iteration in LOTOS
- mutually recursive processes are also allowed; for example:

```lotos
process FLIP [in_0, in_1, out_0, out_1] : noexit :=
    in_1; FLOP [in_0, in_1, out_0, out_1]
endproc

process FLOP [in_0, in_1, out_0, out_1] : noexit :=
    in_0; FLIP [in_0, in_1, out_0, out_1]
endproc
```

- basic LOTOS operators are:

  - `sequence` event; NEXT_PROCESS
  - `enabling` FIRST_PROCESS >> SECOND_PROCESS
  - `choice` POSSIBILITY [] POTENTIALITY
  - `parallelism` INTER || LEAVED
    - EITHER \$ ALPHABET
    - EXPLICIT [subset] ALPHABET
  - `disabling` HEALTH [\$ ACCIDENT
  - `hiding` hide top_secret in MI5
  - `termination` stop
    - exit

- the possibility of successful termination depends on the success of one process for the choice operator, and all processes for the parallel operators
- non-determinism is expressed when the environment is offered a choice of the same events, or with \$ (which may arise explicitly or due to hiding)
- observation equivalence holds when two processes have the same behaviour, ignoring (finite sequences of) internal events
- behaviour definitions give processes names and parameters

```lotos
process VOID [null] : noexit :=
    null; stop
endproc
```
2.8 Exercises

2.8.1 Synchronisation Tree

• consider letters to correspond to events, and words to correspond to sequences of events
• construct a synchronisation tree which spells the words ‘spa’, ‘stop’, ‘strap’, and ‘strop’
• translate this into LOTOS

2.8.2 One-Shot Dictionary

• specify the following application, which has been developed to translate English words into both French and German
• unfortunately, owing to shortage of development funds, the only words which are currently translated are ‘cat’ (‘chat’, ‘Katze’) and ‘dog’ (‘chien’, ‘Hund’)
• even more unfortunately, due to severe overloading of the CPU the system manager has decreed that a user may have only one word translated each time the application is used; after this, the application terminates successfully
• however, when the application was built it was designed with parallel processing in mind; the French and German translation functions therefore run concurrently
• finally, the user has option of killing the entire application at any time, resulting in the application aborting

2.8.3 Events

• what is the alphabet of the behaviour expression:

  BUCKET_BRIGADE [from, to]
  >
  BUCKET_KICKED [done]

given the definitions:

  process BUCKET_BRIGADE [input, output] : noexit :=
    input;
    output;
    BUCKET_BRIGADE [input, output]
  endproc

  process BUCKET_KICKED [dead] : noexit :=
    dead;
    stop
  endproc

• after the first event of the behaviour expression:
2.8. EXERCISES

UNGAINLY QUICKSTEP [lento, rapido, errore]

what are its initials? UNGAINLY QUICKSTEP is defined by:

```plaintext
process UNGAINLY QUICKSTEP [slow, quick, stumble] : noexit :=
  slow;
  (slow; quick; quick; exit
   []
   i; stumble; exit
  )
>>
UNGAINLY QUICKSTEP [slow, quick, stumble]
endproc
```

• give a behaviour expression without internal events which is observationally equivalent to:

I dont see; I to; I with him; stop

• give a behaviour expression without internal events which is observationally equivalent to:

hide a in (i; a; B [a]) || (a; C [a])

• give a behaviour expression without internal events which is observationally equivalent to:

(Paris; DANS) || (le; printemps; exit)

2.8.4 LOTOS-Speak

• it is reported that ISO has instructed all national standards organisations to install LOTOS-Speak machines in their headquarters

• these machines accept a 10 ECU coin and then utter a random sentence allowed by the following grammar (see section 1.4.4 for the notation):

```
sentence = noun_phrase verb_phrase adjective_phrase .
noun_phrase = "LOTOS" [*syntax*].
verb_phrase = "seems" | "proves".
adjective_phrase = adverb adjective .
adverb = "really" | "unbelievably".
adjective = "awful" | "wonderful".
```
• after speaking, the machine is willing to accept a new coin and repeat the above

• unfortunately, the machine was not designed from a formal specification and is liable to fail permanently, issuing \texttt{aargh} once

• specify processes whose permissible sequences of events correspond to the sentences these machines can utter

2.9 \textbf{Possible Solutions}

2.9.1 \textbf{Synchronisation Tree}

• the tree shown in Figure 2.3 may be specified as:

\begin{verbatim}
\begin{verbatim}
s;
  (p; a; stop)
  []
(t;
  (o; p; stop)
  []
(r;
  (a; p; stop)
  []
  (o; p; stop)
  )
  )
  )
  )
\end{verbatim}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{synchronisation_tree}
\caption{Possible Synchronisation Tree}
\end{figure}

2.9.2 \textbf{One-Shot Dictionary}

• a possible solution is:

\begin{verbatim}
( ()
\end{verbatim}
2.9. **POSSIBLE SOLUTIONS**

```plaintext
(cat; chat; exit)
[]
(dog; chien; exit)
)
[cat, dog]
(
(cat; Katze; exit)
[]
(dog; Hund; exit)
)
)
[>
(kill; stop)
```

- this is divided up by language (French translation and German translation), but could also be divided up by word ('cat' translation and 'dog' translation)

### 2.9.3 Events

- from, to, done
- lento, i
- I_dont_see; to; with_him; stop
- hide a in ( B [a] || C [a] )
- ( (Paris; DANS) || (le; printemps; exit) ) [] (printemps; stop)

### 2.9.4 LOTOS-Speak

```plaintext
process MACHINE
[ten_ECU, LOTOS, syntax, seems, proves, really, unbelievably, awful, wonderful, aargh] : noexit :=

SENTENCE
[ten_ECU, LOTOS, syntax, seems, proves, really, unbelievably, awful, wonderful]
[> FAIL [aargh]
endproc

process SENTENCE
[ten_ECU, LOTOS, syntax, seems, proves, really, unbelievably, awful, wonderful] : noexit :=

ten_ECU;
NOUN_PHRASE [LOTOS, syntax]
>>
VERB_PHRASE [seems, proves]
>>
ADJECTIVE_PHRASE [really, unbelievably, awful, wonderful]
>>
SENTENCE
```
[ten_ECU, LOTOS, syntax, seems, proves, really, unbelievably, awful, wonderful]

dendproc

process NOUN PHRASE [LOTOS, syntax] : exit :=

   LOTOS; ( i; syntax; exit) [] exit)

dendproc

process VERB PHRASE [seems, proves] : exit :=

   (i; seems; exit) [] (i; proves; exit)

dendproc

process ADJECTIVE PHRASE
   [really, unbelievably, awful, wonderful] : exit :=

      ADVERB [really, unbelievably]
      >>
      ADJECTIVE [awful, wonderful]

dendproc

process ADVERB [really, unbelievably] : exit :=

   (i; really; exit) [] (i; unbelievably; exit)

dendproc

process ADJECTIVE [awful, wonderful] : exit :=

   (i; awful; exit) [] (i; wonderful; exit)

dendproc

process FAIL [aargh] : noexit :=

   i; aargh; stop

dendproc
Chapter 3

Basic Value Expressions

3.1 Values

- so far, events and event gates have been treated as synonymous; process parameters and results have also
  not been covered

- LOTOS contains a data-typing sub-language called ACT ONE which allows variables and value expressions
  such as:

  Number_of_Hackers
  
suc(suc(suc(0)))

  Concurrency & Synchronisation

- in these course notes, variables names are given initial capitals such as:

  Repeat_Count  Data_Request

- LOTOS variables are mathematical variables: they are not assigned values like a store location; they represent
  the binding of a name to a value

- an expression consists of constants, variables, and operations

- a variable is given a value in a binding occurrence; its scope is generally the behaviour expression which
  follows its binding occurrence

- a sort is a set of values; the following are common sorts:

<table>
<thead>
<tr>
<th>Sort</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bool.Sort</td>
<td>the Booleans ({true, false})</td>
</tr>
<tr>
<td>Int.Sort</td>
<td>the integers (..., -1, 0, +1, ...)</td>
</tr>
<tr>
<td>Nat0.Sort</td>
<td>the non-negative integers (0, 1, ...)</td>
</tr>
<tr>
<td>Nat.Sort</td>
<td>the positive integers (1, 2, ...)</td>
</tr>
</tbody>
</table>

- a variable is given a sort when it is bound, for example:
Bank_balance : Nat_Sort
Liquorice : All_Sort

3.2 Local Definitions

3.2.1 Local Variables

- Local variables may be defined using the local definition operator ‘let ... in’; for example:

  let E : Energy_Sort = M * SQR (C) in SPECIAL_RELATIVITY
  let Salary : Nat_Sort = 50000, Blood_pressure : Nat_Sort = 140 in JOB

- As usual, the sort of a local variable is given in these local definitions following ‘:’
- The scope of such local variables is the behaviour expression which follows ‘in’
- The closest binding determines which value is denoted; for example, assuming decimal notation and arithmetic in the following:

  let X : Int_Sort = 21 in
  let Answer : Int_Sort = X+X in
  let X : Int_Sort = 666 in
  WITMOLTUA (Answer)

  The parameter value for process WITMOLTUA is 42

3.2.2 Local Processes

- Processes may refer to other processes which are purely local to them, defined using ‘where’; for example:

  process LIFE
  [birth, puberty, death, marriage, children] : exit :=

  BIOLOGY [birth, puberty, death]
  [puberty, death]
  FAMILY [puberty, marriage, children, death]

  where

  process BIOLOGY [birth, puberty, death] : exit :=

  birth;
  (puberty; stop)
  [>
  (death; exit)
3.2. LOCAL DEFINITIONS

)  
endproc

process FAMILY
[puberty, marriage, children, death] : exit :=
(
    (puberty;
     (marriage; exit)
     []
     exit
   )
 )
[puberty];
(
    puberty;
    (children; exit)
    []
    exit
   )
 )
[marriage]
(
    (marriage; stop)
[>]
    (death; exit)
   )
 )
endproc
endproc

note that this allows children before marriage and after death; it also excludes infant death in behaviours which do not deadlock!

• the local processes may reference each other and the enclosing process; the enclosing process may reference the local processes

3.2.3 Domains

• events (strictly, gates) and variables may be allowed to range over a set of values; for example:

   Day in [Mon, Tues, Wed, Thurs, Fri, Sat, Sun]

   Prince : Con_Sort
such domains may be used to distribute the following operators over a set:

\[
\text{[] } || \text{ [...]} \]

LOTOS restricts the use of these operators with domains; for example:

\[
\text{choice push, pull : Nat\_Sort } [\text{]} \\
\text{AMPLIFIER (push, pull)}
\]

\[
\text{choice chan in [send, receive], gate in [garden, lych] } [\text{]} \\
\text{BUFFER [chan, gate]} \]

\[
\text{par input in [line1, line2, line3, line4] [control]} \\
\text{MULTIPLEXER [input, control]} \]

only one kind of domain (gates or values) may be used at a time

note that the specified operator indicates what is being distributed, so that the first example above should not be parsed as:

\[
(\text{choice push, pull : Nat\_Sort } ) [\text{]} \\
( \text{AMPLIFIER (push, pull)) } \]

the use of a domain is equivalent to writing out the behaviour expression which follows for each of the event gates or values in the set; for example:

\[
\text{choice event in [write, read] [\text{ ]}} \\
\text{choice B : Bit\_Sort [\text{ ]}} \\
\text{event; MEMORY (B)} \]

is equivalent to:

\[
\text{write; MEMORY (0)} [\text{]} \\
\text{write; MEMORY (1)} [\text{]} \\
\text{read; MEMORY (0)} [\text{]} \\
\text{read; MEMORY (1)} [\text{]} \]
3.3 Guards

- behaviour may be made conditional by using the guard operator ’[…] ->’; for example:

  `[Bonus > 5000] -> ACCEPT_JOB`

- if the conditional expression in the guard evaluates to `true` then the following behaviour is permitted; if the conditional expression evaluates to `false` then the whole behaviour expression is equivalent to `stop`

- this operator may be used to achieve the effect of ‘if...then...else’ or ‘case’, such as might be found in a programming language; for example:

  ```
  ( [Temperature > 0] -> THAW )
  []
  ( [Temperature <= 0] -> FREEZE )
  ```

  ```
  ( [Weather will be Sunny] -> WEAR_T_SHIRT )
  []
  ( [Weather will be Rainy] -> TAKE_UMBRELLA )
  []
  ( [Weather will be Snowy] -> WEAR_ANORAK )
  []
  ( [(Weather won’t be Sunny) and (Weather won’t be Rainy)] and (Weather won’t be Snowy)] -> WEAR_JACKET )
  ```

- a guard may also be an equation between values; for example:

  `core_meltdown = imminent] -> SCRAM`

- note that arithmetic operations (e.g. ‘<’=’) and boolean operations (e.g. ‘and’) are not built into LOTOS; normally, they would be defined in a library, but it is explained later in these course notes how operations can be defined

3.4 Events with Values

3.4.1 Action Denotation

- an action denotation is the event part of an action prefix; so far in these notes, only events which name an event gate have been considered

- in LOTOS, an explicit event properly specifies an event gate and a finite sequence of values; for example:

  ```
  tick
dial ! emergency
statistics ! 36 ! 24 ! 36
  ```
• it is also convenient to specify sets of values; this is called an extended action denotation; for example:

    coord ? X : Int_Sort ? Y : Int_Sort
    colour ? Prim : Primary_Sort

but this is only a short-hand for writing out a choice of event offers with ‘!’ for each value; the last example above is thus equivalent to a choice of:

    colour ! Red
    colour ! Yellow
    colour ! Blue

• both ‘!’ and ‘?’ forms may be mixed; for example:

    calendar ? Month : Month_Sort ! 1986

note that a multi-part event like this is still atomic: its values are not established in a sequence of events; it is therefore not equivalent to:

    calendar ? Month : Month_Sort; calendar ! 1986

• an event may also be followed by a selection predicate – a constraint which restricts the values offered; for example:

    p ? x, y, z : Nat_Sort [SQR (x) + SQR (y) = SQR (z)]

will accept only those triples of values which satisfy the Pythagorean equation; the shorthand notation used here for ‘?’ is equivalent to:

    p ? x : Nat_Sort ? y : Nat_Sort ? z : Nat_Sort

• only the gate name in events is given in constructions such as:

    [gate1, gate2]
    hide g2 in PROC [g1, g2]
    choice gate in [puerta, Tor]
    par gate in [way, crash]
3.4.2 Synchronisation Rules

- only event offers which are identical may synchronise; for example:

  take ! 5

  may synchronise with:

  take ! (2 + 3)

  assuming normal rules for numbers, but not with:

  give ! 5
  take ! 8
  take ! 2 ! 3
  take ! 5 ! 0
  take ! five

- because the ‘?’ form of event is equivalent to a choice between a set of values, ‘?’ may synchronise with ‘!’ if the set of values includes the specifically offered value; for example:

  meeting ? Day : Working_Day_Sort

  may synchronise with:

  meeting ! Tuesday

  but not with:

  meeting ! Sunday

- a further possibility is that both event offers are of the ‘?’ form; in this case, the effect is of an agreement on a value from the intersection of the sets of values (if this is non-empty), provided the values have the same sort for example:

  holiday ? Month : Month_Sort [Is_Summer (Month)]
may synchronise with:

\[
\text{holiday } ? \text{ Mon : Month\_Sort [Has\_31\_Days (Mon)]}
\]

to offer a choice of:

\[
\text{holiday } ! \text{ July}
\]
\[
\text{holiday } ! \text{ August}
\]

whereas:

\[
\text{weight } ? \text{ W : Int\_Sort [W > 0]}
\]

will not synchronise with:

\[
\text{weight } ? \text{ W : Int\_Sort [W < 0]}
\]

### 3.5 Processes with Values

#### 3.5.1 Parameters and Results

- A process may be given *formal parameters* as a list of variables and their sorts; their scope is the behaviour definition of the process.

  For example:

  ```
  \text{process ADDER [input, output] (Base : Nat0\_Sort) : noexit :=}
  \text{input } ? \text{ Offset : Nat0\_Sort;}
  \text{output } ! \text{ Base + Offset;}
  \text{ADDER [input, output] (Base)}
  \text{endproc}
  ```

- A process may also yield a list of results; these are written as a list of sort names *only* following *exit* in the header of the process definition.

  The results of a process are given using *exit* in the body of the process definition; for example:

  ```
  \text{process EMPTY (list : List\_Sort) : exit (Bool\_Sort) :=}
  \text{[eq (list, nil)] -\text{\rightarrow} exit (true)}
  \text{[\_] -\text{\rightarrow} exit (false)}
  \text{endproc}
  ```
the results of a process are made available with the ‘\texttt{>}’ operator using ‘\texttt{accept ... in}’ to match the results list of the process; for example:

\begin{verbatim}
process SUM [ip] (Num : Nat0_ Sort) : exit (Int_ Sort) :=

( [Num = 0] -> exit (0)
 )
( [Num > 0] ->
   ip ? Val : Int_ Sort;
   ( SUM [ip] (Num - 1)
     >>
     accept Total : Int_ Sort in
     exit (Val + Total)
   )
 )
endproc
\end{verbatim}

\subsection{Functionality}

the \textit{functionality} of a process is the set of tuples of results it can produce; for example:

\begin{verbatim}
process DIVIDE [source] (Divisor : Nat_ Sort) : exit (Int_ Sort, Nat0_ Sort) :=

has functionality:

\[
\text{Int}\_\text{Sort} \times \text{Nat}_0\_\text{Sort}
\]

\end{verbatim}

generally speaking, if the behaviour expressions combined by some operator can terminate, their functionality must be the same; for example, \textbf{P1} and \textbf{P2} must have the same functionality in the following cases:

\begin{verbatim}
P1 [] P2
P1 [> P2
P1 || P2
P1 || P2
P1 [...] P2
\end{verbatim}

similarly, the functionality of a process must be the same as for each:

\begin{verbatim}
exit (...)
\end{verbatim}
used in its definition, or for a corresponding:

\[ \text{\texttt{>> accept \ldots in}} \]

- if a process may terminate with no results, just \texttt{exit} is written in its definition header
- if a process cannot terminate, \texttt{noexit} is written in its definition header
- in some circumstances it is convenient to specify that any value of a sort may be a result; for example:

\[
\text{exit (blonde, any Height\_Sort, nubile)}
\]

- this could be used to specify the independent production of parts of a result; for example:

\[
(...) \text{exit (age, any Sex\_Sort)} \]
\[ \| \]
\[
(...) \text{exit (any Nat0\_Sort, sex)}
\]

where \texttt{age} and \texttt{sex} are determined independently by some behaviour expressions in parallel; \texttt{any} is needed because parallel processes which terminate must exit with the same result

### 3.6 Summary

- values are defined and used in LOTOS as follows:

  \[
  \begin{align*}
  \text{\textit{variable}} & \quad \text{Brighton} : \text{Re\_Sort} \\
  \text{\textit{expression}} & \quad (\text{Tom} + \text{Jerry}) - \text{Tweetie\_Pie} \\
  \text{\textit{local variable}} & \quad \text{let Tel} : \text{No\_Sort} = 73171 \textbf{in} \\
  \text{\textit{value domain}} & \quad \text{choice Treaty} : \text{Nat0\_Sort} \ [] \\
  & \quad \text{PEACE (Treaty)} \\
  & \quad \text{guard} [\text{Sentry}] - > \text{BOX (Buck\_Palace)} \\
  \end{align*}
  \]

- the following related concepts are used:

  \[
  \begin{align*}
  \text{\textit{local process}} & \quad \textbf{where process TICK [tock]} \\
  \text{\textit{gate domain}} & \quad \text{choice channel in [English, Bristol]} \ []
  \quad \text{SWIM [channel]} \\
  & \quad \text{par interface in [V24, V35]} \ | \\
  & \quad \text{CONNECT [interface]} \\
  \end{align*}
  \]

- events are specified as an event gate followed by specific values (‘!’) or sets of values (‘?’):

\[
\text{gate ! 13 ? Open : Bool\_Sort}
\]
event offers may synchronise if they are identical; ‘?’ is equivalent to a choice between ‘!’ with specific values

- processes may have parameters and results:

```plaintext
process COMPARE [input] (Ref : Int_Sort) : exit (Bool_Sort, Bool_Sort) :=

input ? Val : Int_Sort;
exit (Val >= Ref, Val <= Ref)

deproc
```

process results are made available with:

```plaintext
COMPARE [data_in] (-40)
>> accept Not_Less, Not_More : Bool_Sort in
   BRANCH [data_out] (Not_Less, Not_More)
```

- the functionality of a process is determined by the sorts of its results; the functionality of terminating behaviour expressions must match in the following cases:

```plaintext
[ ]
[ ]
[ ]
[ ]
[ ]
exit (...) >>> accept ... in
```

### 3.7 Exercises

- give a simpler behaviour expression equivalent to:

```plaintext
( average ? Val : Nat0_Sort [Val < 3]; stop )
```

```plaintext
( choice medium in [ether, average, spiritualist] []
choice Num : Nat0_Sort []
   medium ! Num [Num > 1]; stop )
```
what is the permissible behaviour of:

(gait ! 3; exit (Int)) ; CONFUSING [gait] (1)

and also of:

(gait ! 1; exit (Int)) ; CONFUSING [gait] (2)

where \textbf{Int} is some variable defined to be of type \textbf{Int\_Sort}, and process \textbf{CONFUSING} is defined by:

\begin{verbatim}
process CONFUSING [gate] (Var : Int\_Sort ) : exit (Int\_Sort) :=
  let Var : Int\_Sort = Var - 2 in
  [Var > 0] ->
  gate ? Var : Int\_Sort;
  exit (Var + 2)
  []
  [Var < 0] ->
  gate ? Var : Int\_Sort;
  exit (2 - Var)
endproc
\end{verbatim}

in the following example, \textbf{Next} and \textbf{Break} are constants in some sort representing signal values; after considering a specific example such as:

\begin{verbatim}
FIB [t, r] (1, 1)

| (t ! Next; r ? n : Nat\_Sort;
 t ! Next; r ? n : Nat\_Sort; stop
)
\end{verbatim}

explain in general what the following process may do:

\begin{verbatim}
process FIB [trigger, result] (Seed1, Seed2 : Nat\_Sort) : noexit:=
  ( (trigger ! Next;
  result ! Seed1;
  exit (Seed2, Seed1 + Seed2)
  )
  []
  ( trigger ? Seed1, Seed2 : Nat\_Sort;
\end{verbatim}
3.8 Possible Solutions

- the permitted behaviour is equivalent to:

  average ! 2; stop

- for the first case, the permissible behaviour is:

  gait ! 3; exit (-1)

  provided Int is -1; otherwise it is:

  gait ! 3; stop

  for the second case, the permissible behaviour is only:

  stop

- process FIB may accept a Next request at gate trigger, and output the next number in a Fibonacci series at the gate result

  FIB is initialised with two seed values, but may input new values at the gate trigger; in this case it outputs Break to divide the sequences

  FIB may thus output partial Fibonacci series at the result gate such as:

  1, 1, 2, 3, 5, Break

  and:

  3, 7, Break,
  Break,
  217894, 65932, 283826, 349758, Break
Chapter 4

Data Typing

4.1 Abstract Data Types

- a concrete data type is a representation of structured data within a computer, e.g. a linked list or the layout of a file control block

- an ADT (Abstract Data Type) is an implementation-independent representation of structured data; only the essential characteristics of the data type are specified

- for example, the concrete notion of a book may be described in terms of the quality and colour of its binding and paper, the type of printing ink, the font and size of lettering, etc.

- in abstract terms, a book may be thought of in terms of its components, how these are related, and what may be done to it; its components may be:

  Cover, Title Page, Text, Index

and these components may be further decomposed; for example:

  Title, Author, Date, Publisher

- for Title Page, the relationships between the components may include:

  a Title Page has one Title and one Publisher

  a Title Page may bear several Authors and Dates

  a book is also characterised by what is done to it; for example:

  Print, Read, Plagiarise, Burn
4.2 Type Definitions

• as explained earlier, a sort is a set of data values; a type definition packages up a set of sorts, and specifies the operations on them
  for example:

    type Writer is ... endtype

• do not confuse the name of the type with the names of the sort(s) it defines

• typically, simpler types are built into more complex ones using is to import the definitions in another type; this is called enrichment
  for example:

    type Natural_Number is

        ...

    endtype

    type Boolean is

        ...

    endtype

    type Arithmetic is Natural_Number, Boolean

        ...

    endtype

    type Fraction is Arithmetic

        ...

    endtype

• a type definition contains a number of sections in the following order, each section being optional:

  sorts which gives the names of the sorts defined in the type
  opns which gives the names and functionality of the operations on the sorts
  eqns which gives the equations that the operations satisfy

4.3 Operations

• the operations on a sort are functions, defined by their signature; for example:
4.3. OPERATIONS

increment : Nat_Sort -> Nat_Sort

member : Element_Sort, Set_Sort -> Bool_Sort

An operation may have several sorts as its domain, but has exactly one sort as its codomain.

• An operation must be defined as total, i.e. must be defined for all values in its domain.

• Several operations may have the same signature; for example:

   tomorrow, yesterday : Day_Sort -> Day_Sort

• Operations which have a null domain are constants; for example:

   0 : -> Nat0_Sort

   scarlet, vermilion, crimson : -> Red_Sort

• Operations may be defined as prefix or binary infix; for example:

   is_instructive : Lecture_Sort -> Bool_Sort

   _ * _ : Int_Sort, Int_Sort -> Int_Sort

   Where the ‘_’ stands for a positional parameter; for example:

   pred (0) * succ (0)

• One operation may be chosen as the basic constructor for a sort; there is often a ‘null’ constant which is the simplest value for example:

   add_to_dir : Entry_Sort, Dir_Sort -> Dir_Sort

   empty_dir : -> Dir_Sort

• Operations which extract components of a sort are called selectors; for example:

   name : Entry_Sort -> Alpha_Sort

   address : Entry_Sort -> Alphanumeric_Sort

   telephone : Entry_Sort -> Numeric_Sort
4.4 Equations

4.4.1 Basic Equations

- an operation is specified by giving a list of the *equations* which give the constraints governing it; equations are separated or terminated by ‘;’

- equations may use free variables; for example:

\[
\text{forall } e: \text{Entry\_Sort}, d: \text{Dir\_Sort}
\]

- for the previous example of a telephone directory, the operation:

\[
\text{length\_of\_dir} : \text{Dir\_Sort} \rightarrow \text{Nat\_Sort}
\]

might have equations:

\[
\begin{align*}
\text{length\_of\_dir (empty\_dir)} & = 0; \\
\text{length\_of\_dir (add\_to\_dir (e, d))} & = \text{length\_of\_dir (d)} + 1
\end{align*}
\]

- equations may, if they have the right properties, be regarded as *left-to-right rewrite rules*: if an expression matches the pattern on the left side of an equation it may be replaced by the right side, preserving the same pattern matches

- for example, consider the following operations:

\[
\begin{align*}
\text{nil\_ac} & : \rightarrow \text{Account\_Sort} \\
\text{credit} & : \text{Nat\_Sort, Account\_Sort} \rightarrow \text{Account\_Sort} \\
\text{debit} & : \text{Nat\_Sort, Account\_Sort} \rightarrow \text{Account\_Sort} \\
\text{balance} & : \text{Account\_Sort} \rightarrow \text{Int\_Sort} \\
\text{\_+_\_\_\_\_\_}\_ & : \text{Int\_Sort, Nat\_Sort} \rightarrow \text{Int\_Sort}
\end{align*}
\]

and equations:

\[
\begin{align*}
\text{balance (nil\_ac)} & = 0; \\
\text{balance (credit (n, a))} & = (\text{balance (a)}) + n; \\
\text{balance (debit (n, a))} & = (\text{balance (a)}) - n
\end{align*}
\]

which would require the usual definition of numbers

- for this example, an expression such as:
4.4. **EQUATIONS**

balance (debit (10, credit (5, nil_ ac)))

would be re-written by the third equation above as:

(balance (credit (5, nil_ ac))) - 10

then by the second equation as:

((balance (nil_ ac)) + 5) - 10

and finally by the first equation as:

((0) + 5) - 10

* with definitions for the decimal representation of numbers and arithmetic, this would be further rewritten to the equivalent of -5

4.4.2 **Conditional Equations**

* an equation may be made *conditional* by prefixing it with a boolean condition, like a guard for a behaviour expression; for example:

\[ m \geq n \Rightarrow \max (m, n) = m; \]
\[ m < n \Rightarrow \max (m, n) = n \]

* an equation may also be used instead of a boolean condition; for example:

\[ e1 = e2 \Rightarrow \text{sub\_from\_dir (e1, add\_to\_dir (e2, d)) = d} \]

* as with guards, conditional equations can be used like *if...then...else* or *case* statements

* there is a subtle and important distinction between boolean conditions and equational equality: a boolean condition as a guard is equivalent to writing an equation as a guard for example:

\[ \text{condition} \Rightarrow \text{LHS} = \text{RHS} \]

is equivalent to:
condition = true => LHS = RHS

the equational equality for such a guard holds if the equation can be derived from the equations given in the

• only equality may be derived from the original equations, not inequality; thus, although a boolean condition

  not ((3 * 2) < 7) => wrong = right

may be negated, an equational equality such as:

  (b1 = b2) => b1 exor b2 = true

may not be

• it is therefore usually necessary to define a boolean equality relation (say, ‘==’) between values of a sort;

for example, for the natural numbers:

\[
\begin{align*}
opns
0 & : \rightarrow \text{Nat0\_Sort} \\
succ & : \text{Nat0\_Sort} \rightarrow \text{Nat0\_Sort} \\
_==_ & : \text{Nat0\_Sort}, \text{Nat0\_Sort} \rightarrow \text{Bool\_Sort}
\end{align*}
\]

with the following equations:

\[
\begin{align*}
\text{forall } n, n1, n2 : \text{Nat0\_Sort} \\
0 == 0 & = \text{true}; \\
succ (n) == 0 & = \text{false}; \\
0 == succ (n) & = \text{false}; \\
succ (n1) == succ (n2) & = n1 == n2
\end{align*}
\]

4.5 Overloading

• it is permissible for an operation to be overloaded, i.e. given more than one definition; this is generally used
when the operations are essentially the same but operate on different sorts

for example:
4.5. OVERLOADING

\[
+_+ : \text{Nat0\_Sort, Nat0\_Sort} -> \text{Nat0\_Sort}
\]

\[
+_+ : \text{Matrix\_Sort, Matrix\_Sort} -> \text{Matrix\_Sort}
\]

- sometimes, an overloaded operation may have different domain sorts in more than one definition but the same codomain sort; for example:

  \[
  \text{size : List\_Sort} -> \text{Nat0\_Sort}
  \]

  \[
  \text{size : Dir\_Sort} -> \text{Nat0\_Sort}
  \]

- similarly, the domains may be the same but the codomains may differ; in such cases, the context may indicate which version of the operation is intended for example:

  \[
  \text{forall } e : \text{Entry\_Sort}, d : \text{Dir\_Sort}
  \]

  \[
  \text{size (add\_to\_dir (e, d)) = size (d) + 1}
  \]

- however, the context may not always be sufficient to distinguish which version is meant; in this case, the sort of a value or the result of an operation must be explicitly stated using ‘\text{of}’ for example:

  \[
  5 * ( ((\text{temperature} - 32) \cdot 9) \text{of} \text{Int\_Sort})
  \]

- similarly, for equations it may be logically necessary to use ‘\text{ofsort}’ to indicate the sort of the outermost operation in an equation; for example:

  \[
  \text{ofsort} \text{Bit\_Sort}
  \]

  \[
  \text{forall } b1, b2 : \text{Bit\_Sort}
  \]

  \[
  b1 \text{ implies } b2 = \text{not (b1) or b2}
  \]

  however, LOTOS requires that the sort of an equation be given explicitly

  - \text{ofsort} governs the equations which follow up to the next \text{ofsort}

  - \text{forall} likewise governs the equations which follow; except for the initial, global definitions it must follow \text{ofsort} if it is needed
4.6 Making New Types

4.6.1 Renaming

- a new type may be created simply by renaming an existing one; this may give new names to sorts and operations using 'sortnames...for...' and 'opnnames...for...'

  for example:

  ```
  type Flag is Bool renamedby
  
  sortnames
  
  Flag_Sort for Bool_Sort
  
  opnnames
  
  set for true
  clear for false
  toggle for not
  equiv for ==
  
  endtype
  ```

- note the spelling of opnnames

4.6.2 Parameterisation

- it is frequently necessary to define data types which have the same basic structure and operations but a variety of data elements; LOTOS permits parameterisation of data types to cater for this

- a parameterised data type has formal parameters; for example:

  ```
  type Double_blank is
  
  formalsorts Thing_Sort
  
  formalopns
  
  Z : - → Thing_Sort
  next : Thing_Sort - → Thing_Sort
  _ p_ : Thing_Sort, Thing_Sort - → Thing_Sort
  
  formaleqns
  
  forall T, T1, T2 : Thing_Sort
  
  ofsort Thing_Sort
  
  T p Z = T;
  T1 p (next (T2)) = next (T1 p T2)
  
  opns
  ```
4.6. MAKING NEW TYPES

double : Thing_Sort -> Thing_Sort

eqns

forall T : Thing_Sort

ofsort Thing_Sort

double (T) = T p T

endtype

4.6.3 Actualisation

- a parameterised type is actualised by a specific type to yield a new type
- the formal sorts and operations must be replaced by actual sorts and operations which are compatible with the signatures and equations of the formal ones; for the example given above, a suitable type for actualisation would be:

type Addition is Nat0

opns

_ + _ : Nat0_Sort, Nat0_Sort -> Nat0_Sort

eqns

forall N, N1, N2 : Nat0_Sort

ofsort Nat0_Sort

N + 0 = N;
N1 + succ (N2) = succ (N1 + N2)

endtype

- for this example, the actualisation may be carried out by:

type Duplication is Double_blank actualizedby Addition using

sortnames

Nat0_Sort for Thing_Sort

opnnames

0 for Z
succ for next
+ for p

endtype
4.7 Summary

- abstract data types are implementation-independent representations of structured data:

```plaintext
type Font is Letters ... endtype
```

- an abstract data type is characterised by:
  
  sorts sorts lt_takes_all_Sorts
  operations opns
  Fork : - > Cutlery_Sort
  shiny : Cutlery_Sort - > Bool_Sort
  _picks_up_ : Philosopher_Sort, Cutlery_Sort - > Bool_Sort
  equations eqns
  forall N, N1, N2 : Nat0_Sort
  ofsort Nat0_Sort
  N * 0 = 0;
  N1 * succ (N2) = (N1 * N2) + N1

- equations may be made conditional by prefixing them with a boolean condition or an equation:

  in_Estelle (spec) = > conciseness (spec) = low

  lang (spec) = LOTOS = > writer (spec) = Professor

- operations may be overloaded, which may mean that the sort of a result has to be explicitly stated:

  Coord of X_Coord_Sort
  ofsort French_Sort
  translation (sweetie_pie) = toto

- data types may be renamed:

  Livre renamedby

  sortnames
  Book_Sort for Livre_Sorte

  opnnames
  read for lire
  write for ecrire
4.8. EXERCISES

• parameterised data types are defined using formal parameters:

formalsorts
formalopns
formaleqns

• parameterised data types may be instantiated:

Glue actualized by Paste using

sortnames

Flour_ Sort for Binder_ Sort

opnames

smell for sniff

4.8 Exercises

4.8.1 Improper Fractions

• assume the type Arithmetic defines:
  – sort Nat_Sort
  – operations ‘+’ (addition of natural numbers), ‘*’ (multiplication of natural numbers), and ‘==’ (boolean equality of natural numbers)

• define the type Fractions to have:
  – constructor pair (make a fraction from its numerator and denominator)
  – selectors numer and denom (select the numerator and denominator of a fraction respectively)
  – operations ‘+’ (addition of fractions) and ‘==’ (boolean equality of fractions)

4.8.2 Washing Machine

• a manually-operated machine has operations:
  – load (load the clothes)
  – wash (make the clothes clean)
  – spin (get rid of most of the water from the clothes)
  – tumble (get rid of all the water from the clothes)

• specify this machine, and also the predicates:
  – clean (clothes washed at least once)
  – soaking (clothes washed but not spun or tumble-dried since then)
  – dry (clothes tumble-dried but not washed since then)

• it is presumed that only dirty and dry clothes are initially loaded into the machine
4.8.3 Set of Incrementable Bits

- specify the data type **Bit**:
  - sort **Bit_Sort** with values 0 and 1
  - operations **flip** (binary complement) and ‘==’ (boolean equality)

- specify the data type **Set_with_Incr**, a parameterised set with a special ‘increment’ operation:
  - formal sort **Obj_Sort** (for the elements to be instantiated)
  - formal operations **incr** (increment) and **same** (boolean equality of objects)
  - sort **Set_Sort**
  - operations **empty_set** (the null set), **add_to_set** (add an element to a set), and **in_set** (check for set membership)

- using these definitions, specify the data type **Bit_Set_with_Incr** by actualising the type **Set_with_Incr** with **Bit**

4.9 Possible Solutions

4.9.1 Improper Fractions

type Fractions is Arithmetic

sorts Frac_Sort

opns

pair : Nat_Sort, Nat_Sort -> Frac_Sort
numer, denom : Frac_Sort -> Nat_Sort
_ + _ : Frac_Sort, Frac_Sort -> Frac_Sort
_ == _ : Frac_Sort, Frac_Sort -> Bool_Sort

eqns

forall n, d : Nat_Sort, f1, f2 : Frac_Sort

ofsort Nat_Sort

numer (pair (n, d)) = n;
denom (pair (n, d)) = d

ofsort Frac_Sort

f1 + f2 =
pair ((numer (f1) * denom (f2)) + (numer (f2) * denom (f1)),
denom (f1) * denom (f2))

ofsort Bool_Sort

f1 == f2 =
(numer (f1) * denom (f2)) == (numer (f2) * denom (f1))

endtype
4.9. **POSSIBLE SOLUTIONS**

4.9.2 **Washing Machine**

```plaintext
type Washing_Machine is Bool

sorts

WM_Sort

opns

load : - > WM_Sort
wash, spin, tumble : WM_Sort -> WM_Sort
clean, soaking, dry : WM_Sort -> Bool_Sort

eqns

forall wm : WM_Sort

ofsort Bool_Sort

  clean (load) = false;
  clean (wash (wm)) = true;
  clean (spin (wm)) = clean (wm);
  clean (tumble (wm)) = clean (wm);

  soaking (load) = false;
  soaking (wash (wm)) = true;
  soaking (spin (wm)) = false;
  soaking (tumble (wm)) = false;

  dry (load) = true;
  dry (wash (wm)) = false;
  dry (spin (wm)) = dry (wm);
  dry (tumble (wm)) = true

endtype
```

4.9.3 **Set of Incrementable Bits**

```plaintext
type Bit is Bool

sorts

Bit_Sort

opns

0, 1 : - > Bit_Sort
flip : Bit_Sort -> Bit_Sort
_ == _ : Bit_Sort, Bit_Sort -> Bool_Sort

eqns

ofsort Bit_Sort
```
CHAPTER 4. DATA TYPING

flip (0) = 1;  
flip (1) = 0

ofsort Bool_Sort

0 == 0 = true;  
0 == 1 = false;  
1 == 0 = false;  
1 == 1 = true

dertype

type Set_with_Incr is Bool

formalsorts Obj_Sort

formalopns

incr : Obj_Sort -> Obj_Sort  
_ same _ : Obj_Sort, Obj_Sort -> Bool_Sort

formaleqns

forall o, o1, o2, o3 : Obj_Sort

ofsort Bool_Sort

o same o = true;  
o1 same o2 = o2 same o1;  
o1 same o2, o2 same o3 =>  
o1 same o3 = true

sorts Set_Sort

opns

empty_set : -> Set_Sort  
add_to_set : Obj_Sort, Set_Sort -> Set_Sort  
in_set : Obj_Sort, Set_Sort -> Bool_Sort

eqns

forall o, o1, o2 : Obj_Sort, s : Set_Sort

ofsort Bool_Sort

in_set (o, empty_set) = false;  
o1 same o2 =>  
in_set (o1, add_to_set (o2, s)) = true;  
not (o1 same o2) =>  
in_set (o1, add_to_set (o2, s)) = in_set (o1, s)

ofsort Set_Sort

in_set (o, s) =>  
add_to_set (o, s) = s

(* other axioms are needed to ensure that the order of addition of elements to the set is irrelevant *)

dertype
type Bit_Set_with_Incr is Set_with_Incr actualized by Bit using

sortnames

Bit_Sort for Obj_Sort

opnnames

flip for incr

== for same

endtype
Chapter 5

Putting It Together

5.1 Complete Specifications

• a complete LOTOS specification has a form such as:

    specification IN_SPEC [data] (Limit : Time_Sort) : exit

    behaviour CONTROL [data] (Limit) where

    type Character is
    ...
    endtype (* Character *)

    type Word is Character
    ...
    endtype (* Word *)

    process CONTROL [chan] (Real : Time_Sort) : exit :=
    ...

    where

    process STATION [Crewe, Euston] (T : Train_Set) : exit :=
    ...

    where

    process PLATFORM [P1, P2, P3] : exit :=
    ...
    endproc (* PLATFORM *)

    endproc (* STATION *)

    process CONTROLLER [fat, thin] : exit (Engine_Sort) :=
    ...

    endproc (* CONTROLLER *)
endproc (* CONTROL *)

endspec (* IN_ SPEC *)

- a specification has formal gates parameters and results just like a process; the keyword behaviour (also spelled behavior) gives the behaviour expression which characterises the whole specification
- global type definitions are introduced before behaviour definitions; other type and process definitions come in any order
- type definitions may also be introduced as local, following where
- type definitions in the standard LOTOS library may be included implicitly; for example:

  library

  Set, HexDigit

endlib

- informal comments may be introduced; for example:

  (* This piece of text may be safely ignored *)

- the specification must, of course, be meaningful with all comments deleted; it is a good idea to separate the comments clearly from the formal text

5.2 Semantic Basis

5.2.1 Process Semantics

- the semantics of behaviour expressions is given by action predicates; for example:

  \[ B - g v_1:\cdots:v_n > B' \]

  which postulates that behaviour expression \( B \) may participate in event \( g!v_1:\cdots:v_n \) and then behave like \( B' \)
- for successful termination there is a similar predicate:

  \[ B - !v_1:\cdots:v_n > B' \]

  and for internal action:

  \[ B - \cdot > B' \]

- the semantics of LOTOS operators are given using derivation rules, which have a set of premisses and a conclusion; for example:

  \[
  \frac{G; B - e - > B }{([G] - > B) - e - > B}
  \]

  this says that if boolean guard \( G \) is true, and behaviour expression \( B \) may engage in event \( e \) and then behave like \( B' \), then so may \( ([G] - > B) \)
5.2. SEMANTIC BASIS

• several derivation rules may be required to give the semantics of an operator; for example:
  
  – if \( e \) is not \( \dagger \) (i.e. the \texttt{exit} event):

  \[
  B_1 - e. > B_1' \quad B_1[ > B_1 - e. > B_1' ][ > B_1'
  \]

  – if \( e \) is \( \dagger \) (so that \( B \) must be \texttt{stop}):

  \[
  B_1 - \dagger. > B_1' \quad B_1[ > B_1 - \dagger. > B_1' ][ > B_1'
  \]

  – in all cases:

  \[
  B_2 - e. > B_2' \quad B_1[ > B_2 - e. > B_2' ][ > B_2'
  \]

• given the definitions of the operators and the definition of equivalence or congruence, various \textit{algebraic laws} may be deduced which govern the effect of the operators; for example:

\[
B \parallel \texttt{exit} = B
\]

\[
B_1 \parallel B_2 = B_1 \parallel B_2
\]

\[
B[ \parallel \texttt{stop} = B
\]

\[
(B_1 \parallel B_2') \parallel B_3 = B_1 \parallel (B_2' \parallel B_3)
\]

\[
(e; B_1) [ > B_2 = (e; (B_1 [ > B_2)) \parallel B_3
\]

5.2.2 Data Type Semantics

• the sorts and the operations on them generate an \textit{algebra} of terms; for example:

\[
\text{Empty\_Pot \quad mix (Red, mix (Yellow, Empty\_Pot))}
\]

• the semantics of an \texttt{ACT ONE} data type is given by the \textit{initial algebra}, which makes only those terms equal which are required by the equations to be equal

• for example, a data type may require that:

\[
(c_1 \texttt{IsIn} p) =
\]

\[
mix (c_1, mix (c_2, p)) = mix (c_2, p);
\]

\[
mix (c_1, mix (c_2, p)) = mix (c_2, mix (c_1, p))
\]

the following terms would therefore be made equal in the initial algebra:
mix (Red, mix (Yellow, mix (Red, Empty_Pot)))

mix (Red, mix (Yellow, Empty_Pot))

- initial algebras have a simple operational interpretation using the equations as left-to-right rewrite rules; a 'good' set of equations always re-writes expressions to the yield a canonical ('standard') form in a finite sequence of steps

- executability of a data type definition requires that the right-hand sides of equations are 'simpler' than the left-hand sides; for example:

\[ N_1 + \text{succ} (N_2) = \text{succ} (N_1 + N_2) \]

is a suitable equation because the right-hand side can be further simplified to yield an ultimate expression of the form:

\[ \text{succ} (... (\text{succ} (0))) \]

where there are \( N_1 + N_2 + 1 \) such \text{succ} operations

5.3 Summary

- process semantics are given by:
  - action predicates

\[
B \xrightarrow{gv \; v_1 \ldots v_n} B^\prime \\
B \xrightarrow{jv \; v_1 \ldots v_n} B^\prime \\
B \xrightarrow{\gamma} B^\prime
\]

- derivation rules

\[
\frac{B_i \cdot e \cdot B_{i+1}}{B_i [e] B_{i+1} \cdot B_i^\prime}
\]

- data type semantics are given by the initial algebra which makes only those terms equal which are required to be equal by the equations

- LOTOS is a formal specification language with features for describing static (data type) and dynamic (process) aspects of behaviour

- LOTOS is a general-purpose language for specifying concurrent systems, but was designed to be suitable for specifying OSI services and protocols

- all the examples so far have been of familiar, whimsical or mathematical systems; the following example illustrates how LOTOS can be applied to the specification of communications systems
5.4 A Large Example

5.4.1 Network Service

- the following specification covers data transfer in the OSI Connection-Mode Network Service; although large by comparison with other examples in these notes, only a part of the full specification is included

- the architecture of the OSI Network Service is shown in Figure 5.1

![Network Service Access Points](Figure 5.1: Network Service Access Points)

- the Network Service Provider offers the Network Service to a set of Network Service Users through a set of NSAPs (Network Service Access Points) which can be thought of as abstract interfaces; NSAPs are identified by a Network Address which is unique within the whole Network Service

- in the Connection-Mode Network Service, each Network Service User is attached to one NSAP, as shown in Figure 5.2; multiple connections are allowed through an NSAP

![Network Connection End-Points](Figure 5.2: Network Connection End-Points)

- each connection terminates in an NCEP (Network Connection End-Point); NCEPs are distinguished by a Network Connection End-Point Identifier which is unique within an NSAP

- a Network Service User interacts with a Network Service Provider by means of Network Service Primitives (which can be thought of as abstract interface interactions); two kinds of primitives are identified in this example, as shown in Figure 5.3

5.4.2 Data Transfer

- for this example, data transfer should be specified for just one connection between NSAP A/NCEP a and NSAP B/NCEP b
CHAPTER 5. PUTTING IT TOGETHER

once a connection has been established, data is transferred by means of the following service primitives:

\[
\begin{align*}
\text{N-DATA request} & \quad (\text{NS}_\text{User}_\text{Data}) \\
\text{N-DATA indication} & \quad (\text{NS}_\text{User}_\text{Data})
\end{align*}
\]

\textbf{NS}_\text{User}_\text{Data} is transferred transparently from one User to the other; one or more \textit{octets} (bytes) of data may be sent

- such normal data is subject to \textit{flow-control}, whereby the Service Provider may not accept further data if the connection pipe-line is currently full (the connection capacity may vary and cannot be determined)

- special service primitives are used to send \textit{expedited data}, which will bypass a flow control blockage for normal data:

\[
\begin{align*}
\text{N-EXPEDITED-DATA request} & \quad (\text{NS}_\text{User}_\text{Data}) \\
\text{N-EXPEDITED-DATA indication} & \quad (\text{NS}_\text{User}_\text{Data})
\end{align*}
\]

which allow 1 to 32 octets of data to be sent transparently

- normal data is delivered in the same order as it is transmitted; expedited data is similarly delivered in the same order

- however, expedited data may overtake normal data which is in transit, but it is not guaranteed to do so; the only guarantee given is that normal data which is submitted \textit{after} expedited data will not be delivered \textit{before} it

5.4.3 Suggested Data Type Model

- the following library sorts may be assumed:

  \begin{itemize}
    \item \textbf{Boolean} \hspace{1cm} \textit{Booleans} (sort \textbf{Bool}; constants \texttt{true} and \texttt{false}; operations \texttt{and}, \texttt{or}, and \texttt{not})
    \item \textbf{NaturalNumber} \hspace{1cm} \textit{natural numbers} (sort \textbf{Nat}; operations \texttt{ge} (greater than or equal to), and \texttt{le} (less than or equal to))
    \item \textbf{OctetString} \hspace{1cm} \textit{strings of octets} (sort \textbf{OctetString}; operations \texttt{+} (concatenation of an octet and an octet string) and \texttt{Length} (number of octets))
    \item \textbf{DecNatRepr} \hspace{1cm} \textit{decimal representation of natural numbers} (sort \textbf{DecimalDigit}; constants 0, \ldots, 9; operations \texttt{Dec} (decimal digit to string element) and \texttt{NatNum} (decimal digit string to natural number))
  \end{itemize}

- the following additional types should be defined:

  \begin{itemize}
    \item \textbf{Addr} \hspace{1cm} \textit{Network Addresses, with values \texttt{SAP}_\text{A}, and \texttt{SAP}_\text{B}}
    \item \textbf{Ident} \hspace{1cm} \textit{Network Connection End-Point Identifiers, with values \texttt{CEP}_\text{a}, and \texttt{CEP}_\text{b}}
  \end{itemize}
Data based on OctetString and DecNatRepr, defining Network Service Data Units as octet strings, and defining 1 and 32 as constants with the usual decimal interpretations

- **NSP_Sort** should be defined for Network Service Primitives, with the following operations as constructors:

  NDT_req, NDT_ind

  NEX_req, NEX_ind

- Network Service Primitives correspond to Network Service Objects, which are the pieces of information which are transferred by the underlying medium; a request creates an object, which is later turned into an indication on delivery

- **NSO_Sort** should be defined using NSP_Sort as a basis for Network Service Objects, with operations:

  - req to turn an NSP request into the corresponding NSO
  - ind to turn an NSO into the corresponding NSP indication

- recognisers should also be defined which check the kind of a primitive or object; for example:

  Is_NDT_req (for an NSP)

  Is_NDT (for an NSO)

- Network Connection data transfer behaviour should first be defined using an ordinary queue (‘medium’); a more complex queue which allows expedited data to overtake normal data should then be defined using the operations:

  - delete to delete the first instance of a given object in a queue
  - Is_First to check if an object is the first of its type in a queue
  - Is_Reordered to check if a queue is a valid re-ordering of another

### 5.4.4 Suggested Process Model

- the LOTOS specification should model the Network Connection using processes:

  - NCEP which models the behaviour of one connection end-point
  - NCH which models the behaviour of one half-connection (i.e., direction of transfer)

- the first of these processes should be parameterised by the NSAP address and the NCEP identifier; the second process should be parameterised by the NSAP address and NCEP identifier of both ends

- all communication across the Network Service boundary should be defined to take place at one LOTOS gate called \( n \); invoking a service primitive should correspond to an event of the form:

  \( n ! na ! ni ! nsp \)

  where \( na \) is an NSAP address, \( ni \) is an NCEP identifier, and \( nsp \) is a value constructed for a particular service primitive

- the network connection should be specified by giving the constraints on the behaviour of each end-point and each direction of transfer, as shown in Figure 5.4
5.4.5 Possible Solution

**specification** Subset_Network_Service [n] : noexit

library

Boolean, NaturalNumber, OctetString, DecNatRepr

endlib

type Addr is

  sorts
  Addr_Sort

  opns
  SAP_A, SAP_B : -> Addr_Sort

endtype (* Addr *)

type Ident is

  sorts
  Ident_Sort

  opns
  CEP_a, CEP_b : -> Ident_Sort

endtype (* Ident *)

type Data is OctetString, DecNatRepr

  opns
  1, 32 : -> Nat

  eqns
ofsort Nat
  1 = NatNum (Dec (1));
  32 = NatNum (3 + Dec (2))
endtype (* Data *)

**type** NSP is Boolean, Data

**sorts**
  NSP_Sort

**opns**
  NDT_req, NDT_ind,
  NEX_req, NEX_ind : OctetString -> NSP_Sort
  Is_NDT_req, Is_NDT_ind,
  Is_NEX_req, Is_NEX_ind : NSP_Sort -> Bool
  _eq_ : NSP_Sort, NSP_Sort -> Bool

**eqns**
  **forall** dt, dt1, dt2 : OctetString, nsp : NSP_Sort
  ofsort Bool
  Is_NDT_req (NDT_req (dt)) = true;
  Is_NDT_req (NDT_ind (dt)) = false;
  Is_NDT_req (NEX_req (dt)) = false;
  Is_NDT_req (NEX_ind (dt)) = false;
  Is_NDT_ind (nsp) =>
    Is_NDT_ind (nsp) = false;
  Is_NDT_ind (NDT_ind (dt)) = true;
  Is_NDT_ind (NEX_req (dt)) = false;
  Is_NDT_ind (NEX_ind (dt)) = false;

  Is_NDT_req (nsp) or Is_NDT_ind (nsp) =>
    Is_NEX_req (nsp) = false;
  Is_NEX_req (NEX_req (dt)) = true;
  Is_NEX_req (NEX_ind (dt)) = false;

  Is_NDT_req (nsp) or Is_NDT_ind (nsp) or Is_NEX_req (nsp) =>
    Is_NEX_ind (nsp) = false;
  Is_NEX_ind (NEX_ind (dt)) = true;

  not (Is_NDT_req (nsp)) =>
    NDT_req (dt) eq nsp = false;
  NDT_req (dt1) eq NDT_req (dt2) = dt1 eq dt2;

  not (Is_NDT_ind (nsp)) =>
    NDT_ind (dt) eq nsp = false;
  NDT_ind (dt1) eq NDT_ind (dt2) = dt1 eq dt2;
CHAPTER 5. PUTTING IT TOGETHER

\[
\text{not (Is\_NEX\_req (nsp)) =}
\]
\[
\text{NEX\_req (dt) eq nsp = false;}
\]
\[
\text{NEX\_req (dt1) eq NEX\_req (dt2) = dt1 eq dt2;}
\]

\[
\text{not (Is\_NEX\_ind (nsp)) =}
\]
\[
\text{NEX\_ind (dt) eq nsp = false;}
\]
\[
\text{NEX\_ind (dt1) eq NEX\_ind (dt2) = dt1 eq dt2}
\]

\text{endtype (* NSP *)}

\text{type NSO is NSP}

\text{sorts}
\[\text{NSO\_Sort}\]

\text{opns}
\[\text{req : NSP\_Sort -\rightarrow NSO\_Sort}\]
\[\text{ind : NSO\_Sort -\rightarrow NSP\_Sort}\]

\[\text{Is\_NDT, Is\_NEX : NSO\_Sort -\rightarrow Bool}\]

\[\text{_eq_ : NSO\_Sort, NSO\_Sort -\rightarrow Bool}\]

\text{eqns}
\text{forall dt : OctetString, nsp : NSP\_Sort, nso1, nso2 : NSO\_Sort}
\text{ofsort NSP\_Sort}
\[\text{ind (req (NDT\_req (dt)))) = NDT\_ind (dt);}\]
\[\text{ind (req (NDT\_ind (dt)))) = NDT\_ind (dt);}\]
\[\text{ind (req (NEX\_req (dt)))) = NEX\_ind (dt);}\]
\[\text{ind (req (NEX\_ind (dt)))) = NEX\_ind (dt);}\]

\text{ofsort Bool}
\[\text{Is\_NDT (req (nsp)) =}\]
\[\text{Is\_NDT\_req (nsp) or Is\_NDT\_ind (nsp);}\]
\[\text{Is\_NEX (req (nsp)) =}\]
\[\text{Is\_NEX\_req (nsp) or Is\_NEX\_ind (nsp);}\]
\[\text{nso1 eq nso2 = ind (nso1) eq ind (nso2)}\]

\text{endtype (* NSO *)}

\text{type Basic\_Medium is NSO}

\text{sorts}
\[\text{Medium\_Sort}\]

\text{opns}
\[\text{empty : -\rightarrow Medium\_Sort}\]
\[\text{_ app _ : NSO\_Sort, Medium\_Sort -\rightarrow Medium\_Sort}\]
\[\text{_ pre _ : Medium\_Sort, NSO\_Sort -\rightarrow Medium\_Sort}\]
5.4. A LARGE EXAMPLE

_eq_ : Medium_Sort, Medium_Sort -> Bool

eqns
forall nso, nso1, nso2 : NSO_Sort, ncm, ncm1, ncm2 : Medium_Sort

_ofsort_ Medium_Sort
  nso app empty = empty pre nso;
  (nso1 app (ncm pre nso2)) =
    ((nso1 app ncm) pre nso2)

_ofsort_ Bool
  empty eq empty = true;
  (ncm pre nso) eq empty = false;
  empty eq (ncm pre nso) = false;
  (ncm1 pre nso1) eq (ncm2 pre nso2) =
    (ncm1 eq ncm2) and (nso1 eq nso2)

datatype (* Basic_Medium *)

type Reordering_Medium is Basic_Medium

_opns_
  delete : NSO_Sort, Medium_Sort -> Medium_Sort
  Is_First : NSO_Sort, Medium_Sort -> Bool
  Is_Reordered : Medium_Sort, Medium_Sort -> Bool

eqns
forall nso, nso1, nso2, : NSO_Sort, ncm, ncm1, ncm2 : Medium_Sort

_ofsort_ Medium_Sort
  delete (nso, empty) = empty;
  nso1 eq nso2 =>
    delete (nso1, ncm2 pre nso2) = ncm2;
  not (nso1 eq nso2) =>
    delete (nso1, ncm2 pre nso2) =
    delete (nso1, ncm2 pre nso2)

_ofsort_ Bool
  Is_First (nso, empty) = false;
  Is_First (nso1, ncm2 pre nso2) =
    (nso1 eq nso2)
    or
    (
      (Is_NEX (nso1) and Is_NDT (nso2))
      or
      (Is_NDT (nso1) and Is_NEX (nso2))
    )
    and
    Is_First (nso1, ncm2)
);
Is_Reordered (empty, empty) = true;
Is_Reordered (empty, ncm pre nso) = false;
Is_Reordered (ncm pre nso, empty) = false;
Is_Reordered (ncm1 pre nso1, ncm2 pre nso2) =
  
  (nso1 eq nso2)
  and
  Is_Reordered (ncm1, ncm2)
)

or

  (Is_NEX (nso1) and Is_NDT (nso2))

and

  (Is_First (nso1, ncm2)
  and
  Is_Reordered (ncm1, delete (nso1, ncm2) pre nso2))
)

endtype (* Reordering_Medium *)

behaviour

  (NCEP [n] (SAP_A, CEP_a)
    ||| NCEP [n] (SAP_B, CEP_b)
  )

[n]
  (NCH [n] (SAP_A, SAP_B, CEP_a, CEP_b, empty)
    ||| NCH [n] (SAP_B, SAP_A, CEP_b, CEP_a, empty)
  )

where

process NCEP
  [n] (na : Addr_Sort, ni : Ident_Sort) : noexit :=

  (choice dt : OctetString []
    
    n ! na ! ni ! NDT_req (dt) [Length (dt) ge 1];
    exit
      []
    n ! na ! ni ! NDT_ind (dt);
    exit
      []
  )
5.4. A LARGE EXAMPLE

```
( (n ! na ! ni ! NEX_req (dt)
   [(Length (dt) ge 1) and (Length (dt) le 32)];
   exit []
   n ! na ! ni ! NEX_ind (dt);
   exit )
)

NCEP [n] (na, ni)
```

```
endproc (* NCEP * )
```

```
process NCH
[n] (naX, naY : Addr_Sort, niX, niY : Ident_Sort, ncm1 : Medium_Sort) :
  noexit :=
   ( ( (choice dt : OctetString, ncm2 : Medium_Sort []
          ( n ! naX ! niX ! NDT_req (dt);
            exit (req (NDT_req (dt)) app ncm1)
          []
            [ncm1 eq (ncm2 pre req (NDT_req (dt)))] ->
             n ! naY ! niY ! NDT_ind (dt);
            exit (ncm2)
          )
       )
   )
   ( choice dt : OctetString, ncm2 : Medium_Sort []
       ( n ! naX ! niX ! NEX_req (dt);
         exit (req (NEX_req (dt)) app ncm1)
       []
         [ncm1 eq (ncm2 pre req (NEX_req (dt)))] ->
           n ! naY ! niY ! NEX_ind (dt);
         exit (ncm2)
       )
   )
)

accept ncm2 : Medium_Sort in
  ( choice ncm3 : Medium_Sort []
      [Is_Reordered (ncm3, ncm2)] ->
  )
```

i:
 NCH [n] (naxx, nay, ninX, niY, ncm3)
  }
endproc (* NCH *)

endspec (* Subset Network Service *)
Appendix A
Further Reading


Compound nouns in the index are generally entered under the main noun. Entries for acronyms are indexed under the acronym, but are cross-referenced from the expansion of the acronym. Non-alphabetic entries in the index are sorted according to their ASCII code. The page number of a main entry or definition is given in bold, while the page number of an example is given in italics. Most main entries also give an example.

! (event value), 31
   matching with !, 33
   matching with ?, 33
(*...*) (comment), 58
,. (list separator), 13, 20, 29, 42, 57
- > (function), 42
- > (guard), 31
: (type), 20, 42, 57
:= (defined as), 20
; (action prefix), 11
; (equation separator), 44
= (equation), 44
=> (equation condition), 45
>> (enabling), 14
   and exit, 18, 35
? (set of event values), 32
   matching with !, 33
   matching with ?, 33
[...](gate list), 13, 20, 29
[...] (guard), 31, 32
[> (disabling), 15
[] (choice), 11
_ (positional parameter), 43
[::] (partial synchronisation), 13
| (full synchronisation), 13
\ (interleaving), 12

Abstract Data Type, see ADT
accept, 35
ACT ONE, 2, 27, 59
action
   denotation, 31
   extended denotation, 32
   predicate, 58
   prefix, 11
actualisation, of type, 49
actualizedby, 49

ADT, 41
   see also type
algebra
   initial, 59
   of data types, 59
   operator laws, 59
alphabet, of process, 5
any, 36
assertion, 8
associativity, 8
axiom, 8
behavior, 58
behaviour, 58
behaviour definition, 20
behaviour expression, 4
   see also semantics
   Bool Sort (booleans), 27

Calculus of Communicating Systems, see CCS
cartesian product, 6
case, see guard
CCITT, 2
CCS, 2
choice, see []

choice, 30
   use of gate name, 32
codomain, 7
comment, see (*...*)
Communicating Sequential Processes, see CSP
commutativity, 8
concurrency, see parallel
congruence, 18, 59
observational, 19
constant, 27, 43
constraint
   using parallelism, 13
constructor, of sort, 43

course
  background, 6
  objectives, 1
  scope, 1

CSP, 2

; 13

deadlock, 5, 13

derivation rule, 8, 58

disabling, see [ >
domain, 7, 29

efficiency, 6

enabling, see >>

endlib, 58

endproc, 20

endspec, 57

endtype, 42

enrichment, of type, 42

environment, 4

eqns, 42

equality
  boolean vs. equational, 45
  vs. inequality, 46

equation, 44, 59
  see also eqns
  see also formaleqns

as condition, 45

as guard, 45

conditional, 45

free variable, 44
  see also rewrite rule

equivalence, 18, 59

observational, 19

ESTELLE, 2, 3

event, 4, 31
  kinds, 4
  naming convention, 5
  offer, 4, 16
  set of values, 32
  synchronisation, 33
  value, 31

examples
  adder, 34
  addition, 49
  comparator, 37
  confusing scopes, 38
  double blank, 48
  duplication, 49
  empty list checker, 34
  Fibonacci series, 38
  flag, 48
  life, 28
  list summation, 35
  LOTOS-Speak, 25

network service, 61–70

quickstep, 23

exercises
  events, 22, 25
  expressions, 37, 39
  improper fractions, 51, 52
  LOTOS-Speak, 23, 25
  one-shot dictionary, 22, 24
  set of incrementable bits, 52, 53–55
  synchronisation tree, 22, 24
  washing machine, 51, 53

exit, 57

exit, 13, 20, 34
  and >>, 18, 35

expression, 27

Extended Finite State-machine Language, see ESTELLE

FDT, 2

ESTELLE, 2

history, 2

LOTOS, 2

purpose, 2

SDL, 2

standardisation, 2

sub-groups, 2

for, 48

forall, 47

Formal Description Technique, see FDT

formaleqns, 48

formalopns, 48

formalsorts, 48

function, 7

  codomain, 7
  domain, 7
  in LOTOS, 7
  partial, 8
  total, 8

functionality
  of >>, 36
  of accept, 36
  of choice, 35
  of disabling, 35
  of exit, 35
  of parallel, 35
  of process, 35

gate, 4, 31, 32

gate list, see [...]

grammar, 8

guard
  see also ->
  see also =>
  see also [...]

using equation, 45
hide, 17
  use of gate name, 32
i, 17
  from exit and >>, 18
if...then...else, see guard
in, 17, 28, 29, 35
inaction, 13
inequality
  vs. equality, 46
inference rule, 8
infinite chatter, 19
initials, of behaviour, 5
  i, 13
interaction point, 4
interleaving, see |||
internal event, see i
International Standardisation Organisation, see ISO
International Telephone and Telegraph Consultative
  Committee, see CCITT
Int_Sort (integers), 27
is, 42
ISO, 1
iteration, see recursion

Language Of Temporal Ordering Specification, see LOTOS

let, 28
library, 58
library, of data types, 31
livelock, 5, 19
loop, see recursion
LOTOS
  2
    application, 3
data typing, 5
development, 2
  see also event
    features, 3
  see also library
    nature, 3
    operation, 7
  see also operation
    operators, 5
    recursion, 6
    related languages, 3
    semantics, 58
    standardisation, 2
    syntax, 8
timing, 5
variable, 6
  vs. CCS, 4
vs. CLEAR, 3
vs. CSP, 3
vs. OBJ, 3
vs. programming, 6
member, of set, 6
Nat0_Sort (non-negative integers), 27
Nat_Sort (positive integers), 27
noexit, 20, 36, 57
non-determinism
  through choice, 16
  through internal event, 18
occurrence, binding of variable, 27
of, 47
ofsort, 47
Open Systems Interconnection, see OSI
operation, 27, 42
  see also formalopns
    see also opnames
    see also opns
      binary infix, 43
      built-in, 31
      see also constant
      see also constructor
      see also expression
      see also overloading
      prefix, 43
      see also recogniser
      see also selector
      see also signature
total in LOTOS, 43

opnames, 48
opns, 42
OSI, 1
  problems, 1
over-specification, 6
overloading, of operation, 46

par, 30
  use of gate name, 32
parallel
  for constraints, 13
  full synchronisation, see |||
  interleaving, see |||
  partial synchronisation, see [::::]
  use of gate name, 32
parameter, 34
  actual, 20, 49
  formal, 20, 48
parameterisation, 48
  see also parameter
powerset, 7
process, 4
  alphabet, 5
  see also functionality
  initials, 5
  instantiation, 20
  local, 28
  naming convention, 4
order of definition, 58

*see also* parameter

*see also* result

scope, 29

**process. 20**

recogniser, of sort, 63

recursion, 6, 20

mutual, 21

tail, 21

relation, 7

**renamedby. 48**

renaming, 48

result, 20, 34

rewrite rule, 60

left-to-right, 44

scope, of variable, 27, 28

**SDL. 2, 3**

selection predicate, 32

selector, of sort, 43

semantics

*see also* action predicate

*see also* derivation rule

of behaviour, 58

of data types, 59

sequence

*see also* **>>, **

*see also* :

set, 6

*see also* cartesian product

equality, 6

*see also* member

*see also* powerset

signature, 42

sort, 27, 42

*see also* formalsorts

*see also* sortnames

*see also* sorts

sortnames, 48

sorts, 42

specification, 57

*see also* parameter

*see also* result

**specification. 57**

Specification and Description Language, *see SDL*

**stop. 13**

symbol

non-terminal, 8

terminal, 8

synchronisation

*see also* event

*see also* parallel

tree, 5

syntax, 8

temporal ordering, 5

termination

of choice, 14

of disabling, 15

of enabling, 15

of parallel, 14

of sequence, 14

successful, 13

unsuccessful, 13

type

abstract data, *see ADT*

*see also* actualisation

concrete data, 41

definition, 42

*see also* enrichment

*see also* operation

order of definition, 58

*see also* parameterisation

*see also* renaming, of type

*see also* semantics

*see also* sort

*see also* variable

**type. 42**

**using. 49**

variable, 6, 27

*see also* constant

*see also* expression

free in equation, 44

local, 28

naming convention, 27

*see also* occurrence

*see also* scope

*see also* sort

**where. 28, 58**