SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 22(12), 1069–1098 (DECEMBER 1992)

# An Exercise in Formalizing the Description of a Concurrent System

D. W. BUSTARD
*Department of Computing Science, University of Ulster, Coleraine, BT52 ISA, U.K.*

M. T. NORRIS
*British Telecom Research Labs, Martlesham Heath, Ipswich, IP5 7RE, U.K.*

R. A. ORR
*BT Development & Procurement, 7 Caledon Street, Glasgow, G12 9BY, U.K.*

AND

A. C. WINSTANLEY
*Department of Computing Science, University of Ulster, Jordanstown, BT37 OQB, U.K.*

**SUMMARY**

**LOTOS is one of the most recent formal description languages to appear and one of very few with a standard definition. It has both a process algebra and an abstract data-type component, and these facilities are used in combination to describe the behaviour of concurrent systems. The purpose of this paper is to examine, in a tutorial style, what is involved in constructing and taking benefit from such descriptions. The presentation is illustrated through the development of two formal descriptions for the children's game of pass-the-parcel. These descriptions and a concise summary of the main features of LOTOS are given as appendices. Many of the points made in the paper apply equally well to other process-oriented languages such as CCS and CSP.**

KEY WORDS Formal description Specification styles Concurrency Process algebra Abstract data types LOTOS

## INTRODUCTION

The trend in system development over recent years has been towards the greater use of formality in the expression of system specification and design. The advantages of such an approach are well understood.[1,2] In particular, by constructing a formal, mathematical description of a system, ambiguity is removed, the system described can be analysed before it is built and once constructed can be verified against the formal description. Also, it is possible to refine a formal description into an implementation through a series of correctness-preserving transformations.[3,4]

The purpose of this paper is to illustrate what, in practice, is involved in using a formal description technique in the particular case of describing a concurrent system in the ISO defined language LOTOS .[5,6] The discussion is based around the development of two formal descriptions of the children's game of pass-the-parcel.[7] This

0038–0644/92/121069–30$20.00

*Received 9 May 1990*
*Revised 4 May 1991 and August 1992*

example has been chosen because the 'system' concerned is readily understood and yet is sufficiently complex to suggest how computing applications, such as communication protocols, might be described. The pass-the-parcel game is defined informally in the first section of the paper. The second section gives a short introduction to LOTOS, followed by sections that present and assess two alternative styles of LOTOS use. The paper concludes with a very brief discussion of how programming representations can be derived from a LOTOS description and how LOTOS descriptions can be compared formally for equivalence.

## PASS-THE-PARCEL: NATURAL-LANGUAGE DESCRIPTION

It is usually convenient to develop an informal description of a system in natural language before considering its formal definition. For example, a description of the game of pass-the-parcel might be constructed initially as follows:

1. Children seated in a circle pass a parcel from one to another as music is played.
2. The parcel is wrapped in several layers of paper.
3. The music stops from time to time.
4. When the music stops the child holding the parcel unwraps one layer of paper.
5. If the present is uncovered by removing a layer of paper the game terminates; otherwise the music is restarted and the parcel circulated once again.
6. The game is begun by an adult who passes the parcel to one of the children.
7. The parcel is circulated in a clockwise direction.
8. If two children have a hand on the parcel when the music stops, the child receiving the parcel is assumed to have possession.
9. The game may be interrupted at any time by an adult calling everyone to tea.

Such natural-language descriptions are often criticized for their ambiguity and general lack of precision,[2] and the above description is no exception! Some of the faults it contains are identified later, but it may help to spend a few moments trying to spot the problems at this stage.

One way to detect faults in a description is to turn its statements into a form amenable to mathematical analysis—that is, to construct a *formal model* of the system. Two formal models of the pass-the-parcel game are developed here. The purpose of such models is to clarify and make precise selected key areas of a system description. It is usually the functional aspects of a system that are described formally with non-functional information left in natural language. Essentially, functional requirements relate to what a system must do while non-functional requirements identify constraints on an implementation.[2] For example, in the case of the pass-the-parcel game the functional requirements are the rules of the game, and the non-functional requirements would include a stipulation that the game should be enjoyed by those playing it.

In what follows it is assumed that a full natural-language description is maintained in parallel with any formal representations. In this way there exists a single coherent system description that is available for general communication with a customer or anyone else who needs to understand the system.

## LOTOS: A FORMAL DESCRIPTION LANGUAGE

There are a number of ways in which a formal description might be constructed. One approach is to use variables to represent the *state* of a system and then define system functions in terms of their effect on, or use of, these state variables.[3,8] In the case of the pass-the-parcel game, for example, the state might be represented by three variables:

1. an indication of whether or not music is playing
2. the number of layers of paper on the parcel
3. the identity of the person holding the parcel.

System functions would include operations to stop and start the music and to pass the parcel from one child to another. State-based modelling is mostly used for describing sequential systems.

Another approach to formal modelling is to describe a system in terms of *abstract data types*,[9–11] which identify functions for data manipulation and give them meaning through a set of equations. The equations define interrelationships among the functions. For the pass-the-parcel game the only data item is the parcel itself. A data type representation of the parcel is given later in this paper. The data functions are state-less in that all their inputs and outputs are parameters to the functions. This means that each function is self-contained and therefore can be performed in parallel with any other function. Hence data types can be used to describe aspects of concurrent systems.

One of the most common approaches to the description of concurrent systems is to concentrate on defining system behaviour, usually in terms of the *actions* or *events* involved and the constraints on their order of occurrence.[12,13] For example, in the pass-the-parcel game, events include the stopping of the music and the removal of a layer of paper from the parcel. When describing the game, these two events would be constrained to follow each other. Similar temporal ordering can be defined for other game activities.

The LOTOS language[5,6] supports both a data type and an event ordering approach to system description. More specifically, LOTOS, which is an acronym for *Language of Temporal Ordering Specification*, provides

(a) an *abstract data type* (ADT) formalism, based on ACT ONE,[14] that can be used to describe system data and the operations performed on it
(b) a *process algebra* formalism, based on CCS (Calculus of Communicating Systems)[12] and CSP (Communicating Sequential Processes) 13 that can be used to describe system behaviour.

LOTOS was developed to meet the specific needs of the OSI (Open System Interconnection) community who use the language for the specification of protocols and services.[15–17] Its definition is sufficiently general, however, to allow it to be applied to most system descriptions. A concise summary of the language is given in Appendix I, but its features are explained, where necessary, in the discussion that follows. Other introductions to the language may be found in References 5 and 6.

In this paper most emphasis is given to the behavioral approach to formal description. LOTOS is intended to be used to specify systems at a variety of levels of abstraction from high-level abstract specifications to detailed models of system components. This flexibility means that different LOTOS specifications will be

written in different styles depending on their purpose. Various specification styles have been discussed in the literature.[19,20] These tend to fall into two broad categories: *event-based* and *object-based* styles. These styles are not mutually exclusive, and a particular specification may contain different parts constructed in different ways.

With the event-based style, most emphasis is placed on defining the temporal ordering of system events. With the object-based style the underlying concern is still to define an event order but the description is structured as a collection of components (processes and data types) corresponding to areas of responsibility in the system being defined. The next two sections explain and illustrate each of these descriptive styles in turn.

## FORMAL DESCRIPTION: THE EVENT-BASED STYLE

The event-based style of formal description essentially seeks to define the behaviour of a system in terms of the events in which it participates and the constraints on their order of occurrence. In the pass-the-parcel game, for example, the events of interest include the starting and stopping of music, the transfer of a parcel from one child to another and the interrupting call to tea. Such events can, in general, be readily deduced from the natural-language description of a system. Specific examples of event-based styles [20] include the *constraint-oriented* style, which uses processes composed in parallel to isolate the different constraints on permissible event sequences, and the *slice style* [21] which models sequences of permissible interactions propagating through the components of a reactive system. The *monolithic style* is one that contains no internal process structure but defines explicitly the permissible sequences of events. This style is rarely used for complete specification construction due to its lack of structure, but it can be a useful way to envisage small process definitions.

If the number of system events present is sufficiently small and their interrelationship sufficiently obvious then a formal description can be produced directly. If not, then it is usually helpful to construct a diagram outlining the relationships involved. These diagrams may be informal, but formal notations are available. Concurrent systems may, for example, be represented by Petri nets [22] or communicating finite-state machines. [23] In the case of the pass-the-parcel game, for instance, the behaviour may be described by a single finite-state machine as shown in Figure 1.

In the diagram the circles represent possible system states and the arcs represent events that cause transitions among those states. All systems have an initial state and at least one final state. The initial state in this case is identified at the top of the diagram and the two possible end states are on the right, distinguished by double circles. The diagram indicates that the call to tea can occur at any time, that the parcel passes while music is playing and that once the music stops a layer of paper is removed from the parcel, after which the game either terminates (because the present has been found) or the parcel is returned to circulation.

Some attempts have been made to define a graphical version of LOTOS [24] to allow specifications to be built in a diagrammatic form. It is also possible to define formal transformations between descriptions, such as a state-transition diagram and a LOTOS definition [25] to ensure consistency when transferring between representations.

The description of the game in LOTOS in an event-based style first identifies the events involved:

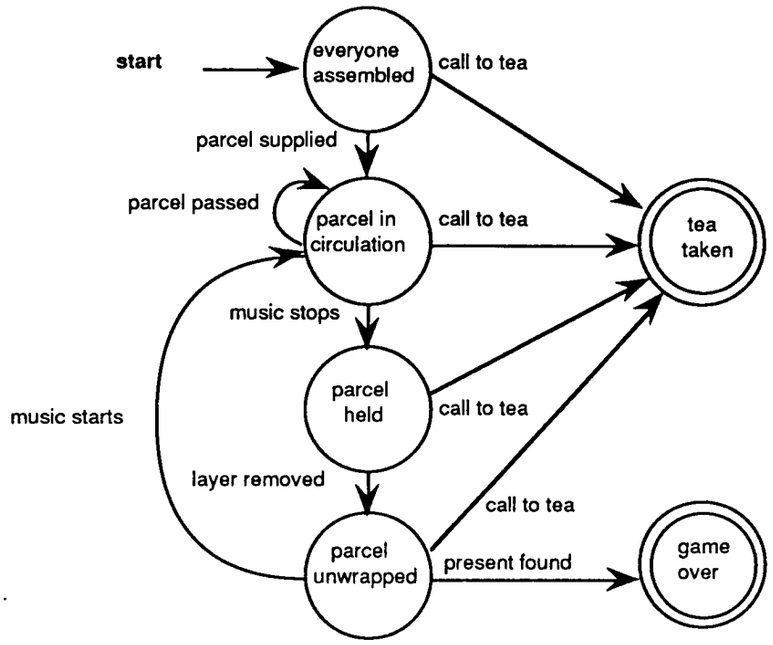


*Figure 1. State-transition diagram for the pass-the-parcel game*

```
specification PassTheParcel [ParcelSupplied, ParcelPassed, MusicStarts,
    MusicStops, LayerRemoved, PresentFound, CallToTea]: exit
  behaviour
    ...
endspec
```

The events are named as *event gates* in the specification heading. The word exit, at the end of the heading, indicates that the system described terminates. The remainder of the specification defines the behaviour of the system in terms of the named events. This can be achieved by first describing the main concern, that of passing the parcel, and then constraining that activity appropriately. The passing of the parcel is defined by a *process,* and other processes are linked to it to implement the constraints. Processes are structured in a hierarchical fashion with the overall specification itself being a process. Apart from minor (irritating!) syntactic differences a LOTOS process has the same structure as the specification. For example, the main activity of passing the parcel might be expressed as follows:

```
1. process Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved,
   PresentFound]: exit :=
2. ( ParcelPassed;
3.       Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved, PresentFound])
```

```
4.  []
5.  ( MusicStops;
6.        LayerRemoved;
7.        (( MusicStarts;
8.           Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved,
             PresentFound])
9.        []
10.       ( PresentFound;
11.          exit)))
12.   endproc
```

This representation is consistent with the description given in Figure 1 except that the calltotea event is not yet specified. It will be introduced shortly. Note the following general points about the LOTOS notation shown:

1. Events (or actions, in general) that occur in sequence are combined using a semicolon—the *action prefix* operator (e.g. lines 2, 5, 6, 7 and 10).
2. If at any time the next event to occur is indeterminate, the possible events and subsequent actions are combined with a *choice* operator producing an expression of the form b1 [] b2 (e.g. lines 4 and 9).
3. Looping behaviour is represented by recursive definitions (e.g. lines 3 and 8).
4. The end of a terminating process is marked by an exit (e.g. line 11).

A more precise definition of these and other operators is given in Appendix I.

Thus, the circulation of the parcel is described as a choice between the passing of the parcel ( ParcelPassed ) and a response to the music stopping ( MusicStops ). In the latter case, a layer of paper is removed from the parcel ( LayerRemoved ) after which the music starts again ( MusicStarts ) or the game terminates ( PresentFound ), depending on whether or not paper remains on the parcel. Once the parcel is passed, or the music restarted, the system returns to its original state, indicated by a recursive instantiation of Circulate. Notice that with this approach to specification no distinction need be made between describing sequential and concurrent systems. Events are simply ordered, and where the order is partial there is potential for concurrent behaviour. Note also that processes are being used to structure constraints rather than identify concurrent behaviour.

The constraints that the parcel has first to arrive and that a call to tea can occur at any time can now be added to the basic description of parcel passing. The constraints can be defined by processes linked to the Circulate process using appropriate operators. For example, a Prepare Parcel process may be linked to the Circulate process using an *enable* operator >> to indicate that the Parcel Supplied event must occur before the circulation of the parcel can begin:

```
PrepareParcel [ParcelSupplied]>>
Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved,
PresentFound]
```

The PrepareParcel process simply identifies the ParcelSupplied event, thus:

```
process PrepareParcel [ParcelSupplied]: exit: =
   ParcelSupplied; exit
endproc
```

Similarly, the process

```
process Tea [CallToTea]: exit :=
  CallToTea; exit
endproc
```

can be linked to the PrepareParcel and Circulate processes using the *disable* operator [> to indicate that the game may be interrupted at any time by a call to tea:

```
(PrepareParcel [...] >> Circulate [...]) [> Tea [CallToTea]
```

In general, the act of building such a model will often reveal problems in a natural-language description. In this case, for example, there is an unspecified assumption that music is playing initially—a fault that has been carried through to the diagrammatic depiction of the system in Figure 1 and to the corresponding LOTOS description given above. In producing either of these representations the problem might be noticed and corrected. The correction would be made to each representation maintained. For example, the natural-language version might be extended with a statement of the form:

10. Music is playing when the parcel is passed to the first child.

The corresponding adjustment to the LOTOS description can be achieved by introducing another constraint process, InitialMusic, thus:

```
(InitialMusic [MusicStarts] >>
  PrepareParcel [ParcelSupplied] >>
    Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved,
    PresentFound])
[> Tea [CallToTea]
where

process InitialMusic [MusicStarts]: exit :=
  MusicStarts; exit
endproc (* Initial Music *)
```

In general, other less obvious problems are uncovered by *animating* the formal description. For LOTOS this means exploring the event sequences that the description permits. [26–29] Conceptually, the set of valid execution paths forms an *action* or *derivation* tree. [6] For example, part of the tree for the LOTOS pass-the-parcel description is shown in Figure 2.

Arcs denote events and nodes represent system states. Multiple arcs from a single node identify a state from which one of several subsequent events may occur. In the pass-the-parcel specification, for example, there are at least two arcs from each node because the CallToTea event can occur at any time. Note that this is an infinite tree because there is no requirement in the LOTOS description that the music should ever stop.

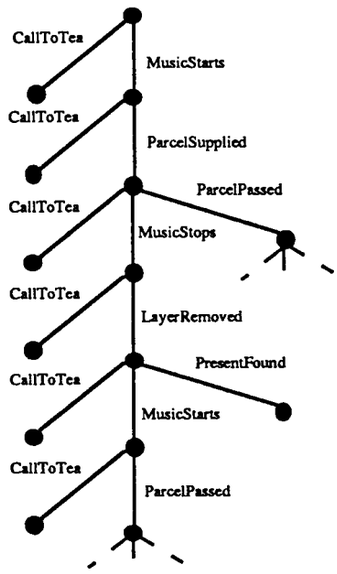


*Figure 2. Partial action tree for the pass-the-parcel game*

In many respects animating a LOTOS description is the same as program testing, and so can be approached in much the same way and with the same objectives.[30] Specifications are usually explored bottom-up by first animating basic processes whose behaviour is defined directly in terms of events; processes whose definitions are derived from the basic processes can then be tackled and analysis proceed in the same way up to the top specification level.

An animation of the full pass-the-parcel specification reveals that the MusicStops event is offered immediately after the MusicStarts event, which means that a child can retain the parcel and remove successive layers of paper until the present is uncovered. In natural-language terms, the constraint to exclude this possibility might be expressed as follows:

11. After removing a layer of paper from a parcel and failing to reveal the present, a child must pass on the parcel immediately the music restarts.

As before, this restriction can be added to the LOTOS description using a constraint process. However, the process required has a structure similar to Circulate so it is preferable to modify Circulate directly by placing the event ParcelPassed immediately after MusicStarts. Following such modifications it is of course necessary to repeat the animation to ensure that the resulting behaviour is as required.

The formal description of the game in an event-based style is now complete and may be found in full in Appendix II. The LOTOS description is relatively short and (for someone experienced in the language) would require little effort to produce. Finding problems of the type illustrated above makes such an exercise well worth while. However, the LOTOS description covers only part of the game definition, ignoring, in particular, the parcel and the individual children. It is therefore tempting

to extend it to include this additional detail. The extension introduces lower-level concerns and so is analogous to refining a specification towards an implementation.

In general, refinement involves converting an abstraction description into versions that are less abstract through a sequence of intermediate steps that add constraints (implementation details) to the definition or take account of further requirements. [4] Each development step can be made in one of two ways:

1. produce a new description and prove it consistent with the preceding version; or
2. transform one description into another by refining aspects of the first according to a set of formal rules.

The second approach is more appealing because any form of proof tends to be difficult and time-consuming. However, in practice, the first approach is often more appropriate because it can be necessary to restructure a formal description as it is refined. This is to be expected, as initial descriptions will focus on requirements, whereas those closer to the implementation will tend to reflect the structure of the system to meet the requirements. The next section illustrates this transition by showing the development of a LOTOS description of the pass-the-parcel game in an object-based style. This description includes details of the behaviour of each individual child and the operations each performs on the parcel. However, as far as possible, the set of events present in the event-based definition is carried across so that the two descriptions can be compared more easily. The problem of ensuring that the two descriptions are consistent is discussed in a later section.

## FORMAL DESCRIPTION: THE OBJECT-BASED STYLE

The object-based style of formal description uses processes and data types to model the recognizable entities in a system. For example, in the pass-the-parcel game the entities are the music, the *adult* who starts the game, the *children* who play the game, the *adult* who calls the children to tea and the *parcel.* This contrasts with the focus on behaviour used to develop the object-based description. Object-based specifications have an internal structure that reflects that of an actual implementation. Specific styles in this category include the *state-oriented* style, [20] which uses state variables passed as value parameters to recursive processes to represent subsystems. Often semi-formal specifications use state-based models of physical devices, and so a LOTOS specification can be produced to correspond with these. This style therefore tends to be much more implementation-oriented. The *resource-oriented* style [20] is even more implementation-specific as the physical resources available for implementation are directly modelled by LOTOS processes. The specifier models the individual behaviour of each component and then composes these to define the overall behaviour of the system.

With the object-based approach it is often convenient to first construct an entity-interaction diagram that identifies the entities and shows those that are interdependent. Figure 3, for example, is a possible entity-interaction diagram for the pass-the-parcel game. In this diagram the adults involved are, for convenience, identified as a *mother* and a *father,* with their tasks assigned arbitrarily. The father waits for the music to start (1) and then passes the parcel (2) to one of the children (3); the children manipulate the parcel (4) in response to the stopping and starting of the music (5); the mother may stop the game at any time (6, 7, 8).

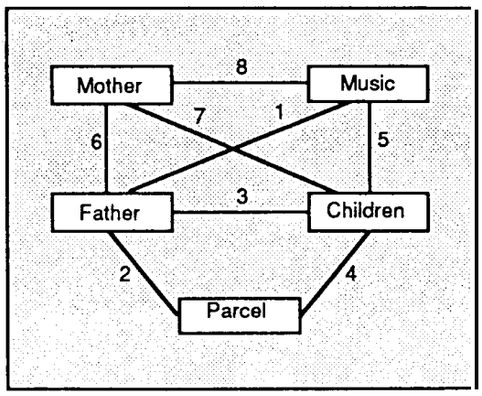


*Figure 3. Entity-interaction diagram for the pass-the-parcel game*

The entities in a system represent potential concurrent behaviour in that each might have autonomy and interact with other entities by agreement. However, it is possible to draw a distinction between *active* and *passive* entities and choose different representations for each. For example, the father, mother, music and children are conceptually *active entities* in the system and so may be represented by processes. The parcel is a *passive entity* and so can be represented as a data type. This classification is somewhat controversial and Milner, in particular, argues that it is neither practical nor desirable to make such a distinction. [31] The parcel can indeed be represented by a LOTOS process. However, its definition as a data type will be given here to illustrate both this approach to formal description and its integration with a process-algebra notation.

In general, a LOTOS specification is made up of

1. A set of *environment parameters* identifying
   (a) named *values* that instantiate the definition
   (b) possible interactions between the system specified and its environment, defined by event gates
   (c) data types in the specification taken from a *data type library.*
2. A hierarchy of *process* definitions describing the intended system behaviour in terms of permitted sequences of *events.*
3. A set of *data-type* definitions describing the data manipulated by the processes.

In an object-based style of description, the events corresponding to process interaction are usually hidden as far as possible. That is, each event is associated as closely as possible with that part of the description in which its use is defined. Thus at the top level of a LOTOS description the only event gates that need to be named are those for external communication. In the case of the pass-the-parcel game, it can be argued that there is external communication with the observer of the game and thus that all events need to be named in the specification heading. Another interpretation is that no events should appear in this position because the system described is self-contained. This latter view will be used here as a contrast to the approach taken in the preceding section. Thus the basic form of the specification would be as follows:

```
specification PassTheParcel (ParcelSize: Nat, Players: Nat): exit
  library NaturalNumber, Boolean endlib
  behaviour . . .
endspec
```

The first line of the specification introduces two value parameters that help to make the definition of the system more general: ParcelSize, denoting the number of layers of paper on the parcel and Players, denoting the number of children playing the game. Both parameters are natural numbers.

The second line of the specification identifies library types that are used: NaturalNumber and Boolean. For natural numbers and Booleans, in common with most data types, a distinction is made between the values, or *sort*, of each type and its overall definition (this will include descriptions of the operations that can be performed on values of that type). Natural numbers have a data-type name NaturalNumber and a sort name Nat. The sort name for the Boolean type is Bool.

The behaviour expression for the game might be as follows:

```
[(Players le succ(0)) or (ParcelSize eq 0)] –> ( exit )
[]
[Players gt succ(0) and (ParcelSize gt 0)] -> (Game (ParcelSize, Players) [>
Mother)
```

This is a guarded-choice expression indicating that the game can only be played if there are at least two children available and at least one layer of paper on the parcel (two restrictions not mentioned explicitly in the natural-language description and which should be added to it). Using the operations defined for NaturalNumber, the value ‘ 1’ is denoted by succ(0), i.e. the successor of zero. If the game can be played, the subsequent behaviour of the system is described by two processes: Game and Mother, where Game describes the playing of the game and Mother describes the action of the adult who calls the children to tea. The relationship between the two processes is defined by the disabling operator that connects them, as in the event-based specification. In this case, however, the interrupting event is internal to the Mother process:

```
process Mother: exit : =
  i; exit
endproc (* Mother *)
```

The action denotation i represents the call to tea, which is followed by a termination exit. The action i is an *internal event* —one in which no other process participates (this serves the same purpose as the $\tau$ event in CCS [12] ). Because of the disabling operator connecting Mother to Game any event in Mother will cause the termination of the Game process.

The definition of the Game process describes the behaviour and interaction of the children, the music and the father who supplies the parcel. The parcel itself may be defined at this point:

```
1   type ParcelType is NaturalNumber, Boolean
2     sorts Parcel
3     opns (* operations *)
4       NewParcel:          Nat      –> Parcel
5       Wrappers Remain: Parcel      –> Bool
6       Unwrap:             Parcel   –> Parcel
7     eqns (* equations *)
8       forall n: Nat
9       ofsort Bool
10        WrappersRemain (NewParcel (0))        = false;
11        WrappersRemain (NewParcel(succ( n))) = true;
12      ofsort Parcel
13        Unwrap (NewParcel(0))          = NewParcel (0);
14        Unwrap (NewParcel(succ( n))) = NewParcel (n);
15  endtype (* ParcelType *)
```

The first line of the definition indicates that ParcelType is based on the definitions of natural numbers and Booleans. The remainder of the definition has three parts:

1. line 2: the identification of a new *sort* — Parcel, a name for the values taken by the type
2. lines 3–6: a set of *operations* (or *functions)* and their parameters (inputs to the left of the arrow, outputs to the right), identifying what can be done with a parcel
3. lines 7–14: a set of *equations* defining properties of those operations—in effect, the equations define what the operations mean.

A parcel is manipulated by three operations: New Parcel, which creates a parcel, Wrappers Remain, which reports whether or not the present has been uncovered, and Unwrap, which removes a layer of paper from the parcel. The properties of the parcel operations are defined by accompanying equations. For example, one equation (line 14) defines the Unwrap operation to be equivalent to the inverse of the successor operation (succ) for natural numbers, and two others (lines 10 and 11) indicate that WrappersRemain is true if the number of wrappers is non-zero. Note that the parcel-type definition makes no statement about the implementation of a parcel, restricting itself entirely to a set of abstract operations for its manipulation.

For the game itself, processes can be used to describe the behaviour of the father, the music and the children. In essence, therefore, the behaviour expression of the game takes the form:

Father || Children || Music

where the parallel bars indicate that the behaviour of these entities is concurrent.

The events shared by the processes must also be identified in the behaviour expression. An event, in general, denotes the synchronization of two or more processes and may involve the communication of one or more data values. In the latter case the interaction is often referred to as a *structured event.* The Music process is required to indicate when the music starts ( MusicStarts ) and stops ( MusicStops ). The Children process accepts the parcel initially ( ParcelSupplied ), and responds to the music stopping ( MusicStops ) and starting ( MusicStarts ). When the music stops a layer

of paper is removed from the parcel (not modelled at this level), after which the game either terminates, if the present has been uncovered ( PresentFound ), or continues if layers of paper remain.

The events on which a process synchronizes are passed as event-gate parameters in the instantiation of that process, and the events shared by particular processes are identified in the parallel operators connecting those processes. For example, assuming that the Children and Father processes synchronize using the event gate Parcel Supplied, their connection would take the following form:

```
Father [MusicStarts, ParcelSupplied] (ParcelSize)
   |[ParcelSupplied]|
Children [ParcelSupplied, MusicStarts, MusicStops, PresentFound] (Players)
```

The Father process also synchronizes with the Music process on the MusicStarts event (music must be playing before the parcel is supplied), and the Children process synchronizes with the Music process on the MusicStarts, MusicStops and PresentFound events:

```
(Father [MusicStarts, ParcelSupplied] (ParcelSize)
   |[ ParcelSupplied]|
Children [ParcelSupplied, MusicStarts, MusicStops, PresentFound] (Players))
|[ MusicStarts, MusicStops, PresentFound]|
Music [MusicStarts, MusicStops, PresentFound]
```

Note that the bracketing arrangements here are equivalent to

```
(A |[x]| B) |[y]| C
```

The parallel operator is binary and associative. Linking groups of processes with a parallel operator requires only *one* process on each side of the operator to synchronize for each connecting event. Multi-way synchronizations can be achieved by naming the same event in more than one parallel operator. Thus, in the above expression, a three-way synchronization will occur for each event that is common to x and y; process C will synchronize with *either* process A or process B on events exclusive to y.

Returning to the pass-the-parcel example, it can be noted that the ParcelSupplied, MusicStarts, MusicStops and PresentFound events, which are local to the Game process, are *hidden* thus:

```
process Game (ParcelSize: Nat, Players: Nat): exit :=
  hide ParcelSupplied, MusicStarts, MusicStops, PresentFound in...
endproc (* Game *)
```

Now consider the expression of the Father, Music and Children processes. The Father process might take the following form:

```
process Father [MusicStarts, ParcelSupplied] (ParcelSize: Nat): exit : =
  MusicStarts; ParcelSupplied ! NewParcel (ParcelSize); exit
endproc (* Father *)
```

The father waits for the music to start and then supplies the parcel, represented by the structured event shown. The corresponding acceptance of the parcel is denoted thus :

```
ParcelSupplied? Parcel: Nat
```

In LOTOS, the ‘ ! ’ operation means that a single value is offered for synchronization, whereas ‘ ? ’ means that a set of values is offered. Synchronization then occurs whenever:

(a) two (or more) processes have *identical* event offers, or
(b) two or more processes have *matching* offers; that is, where the values offered by one process are in the set of acceptable values offered by another—each naming the same event gate; when several values are possible one is selected randomly.

Thus, in LOTOS, the model of communication is one of values being *agreed* among processes as distinct from being transmitted explicitly among them. (Note that operators ‘ ! ’ and ‘ ? ’ should not be confused with those used in CSP, [13] where ‘ ! ’ is read as *output* and ‘ ? ’ as *input. )*

It now remains to consider the definition of the Children and Music processes. The process definitions must be constructed to handle clean termination (rather than deadlock). For example, the Music process might take the following form:

```
process Music [MusicStarts, MusicStops, PresentFound]: exit : =
   MusicStarts;
   MusicStops;
   ((Music [MusicStarts, MusicStops, presentFound])
   []
   (PresentFound; exit ))
endproc (* Music *)
```

Here the behaviour is simply to offer the MusicStarts and the MusicStops events until the present has been found ( PresentFound ) leading to the termination of the process.

The modelling of the, children is considerably more difficult. Assuming that each is represented separately, the following process might be used:

```
process Child [ParcelSupplied, Parcelin, ParcelOut, MusicStarts, MusicStops,
PresentFound] (Parcel From Father: Bool): exit : =
   [ParcelFromFather]-> (* first child accepts parcel from father *)
      (ParcelSupplied ? The Parcel: Parcel;
      ParcelAction [ParcelSupplied, Parcelin, ParcelOut, MusicStarts, MusicStops,
      PresentFound] (TheParcel))
   []
   [not (ParcelFromFather) ]->
      (( PresentFound; exit )
      []
```

```
        (Parcelin ? TheParcel: Parcel;
        ParcelAction [ParcelSupplied, Parcelin, ParcelOut, MusicStarts, MusicStops,
        PresentFound] (The Parcel)))
    ...
endproc (* Child *)
```

A child may either receive the parcel from another child or from the adult who introduces the parcel into the circle. These two cases are distinguished by a Boolean parameter ParcelFromFather, supplied to each instantiation of the Child process. Regardless of the source of the parcel, the subsequent actions performed are identical and defined by the process ParcelAction, thus:

```
process ParcelAction [ParcelSupplied, Parcelin, ParcelOut, MusicStarts,
MusicStops, PresentFound] (The Parcel: Parcel): exit :=
  ( ParcelOut ! The Parcel;
    Child [ParcelSupplied, Parcelin, ParcelOut, MusicStarts, MusicStops,
    PresentFound] (false))
    []
  ( MusicStops;
    i; (* remove layer of paper *)
    ( let UnwrappedParcel: Parcel = Unwrap (TheParcel) in
      ( [Not (WrappersRemain (UnwrappedParcel) )]–>
          (PresentFound; exit )
        []
        [Wrappers Remain (Unwrapped Parcel)]- >
          (MusicStarts;
          ParcelOut ! UnwrappedParcel;
          Child [ParcelSupplied, Parcelin, ParcelOut, MusicStarts, MusicStops,
          PresentFound] (false))
      ) (* selection on WrappersRemain or not WrappersRemain *)
    ) (* scope of Unwrapped Parcel *)
  ) (* MusicStops selection *)
endproc (* ParcelAction *)
```

The parcel supplied as a parameter to this process is passed on to another Child process using the gate ParcelOut. On receiving the parcel a child either passes it on or, if the music has stopped, removes a layer of paper (denoted by an internal event). If, as a result, the present is uncovered the end of the game is reported; otherwise the child waits for the music to start ( MusicStarts ) and then puts the parcel back into circulation.

The final part of the description to consider is the definition of the Children process. This process is far from straightforward because:

(a) the number of Child processes needed is based on a value passed as a parameter to the specification, and
(b) the Child processes share gates that effectively connect them in a circle.

The Child processes may be created by instantiating them one at a time as follows:

```
process Children [ParcelSupplied, MusicStarts, MusicStops, PresentFound]
(Players: Nat): exit : =
  hide FirstLink, NextLink in
    (Child [parcelSupplied, FirstLink, NextLink, MusicStarts, MusicStops,
    PresentFound] (true)
    |[FirstLink, NextLink, PresentFound]|
    StartChild [ParcelSupplied, FirstLink, NextLink, MusicStarts, MusicStops,
    PresentFound] (succ (succ(0)), Players))
    where
    process child . . . endproc (* Child *)
    process StartChild [ParcelSupplied, FirstLink, PreviousLink, MusicStarts,
    MusicStops, PresentFound] (Identity: Nat, Players: Nat): exit :=
      [Identity eq Players]-> (* last child *)
        (Child [parcelSupplied, PreviousLink, FirstLink, MusicStarts, MusicStops,
        PresentFound] (false))
      []
      [Identity lt Players] –> (* child other than the first or last instantiated *)
        ( hide NextLink in
          Child [ParcelSupplied, PreviousLink, NextLink, MusicStarts,
          MusicStops, PresentFound] (false)
          |[NextLink, PresentFound]|
          StartChild [ParcelSupplied, FirstLink, NextLink, MusicStarts,
          MusicStops, PresentFound] (succ (Identity), Players))
    endproc (* StartChild *)
endproc (* Children *)
```

A process StartChild is responsible for (recursively) instantiating the required number of Child processes apart from the first one. The StartChild process is parametrized with:

(a) a gate PreviousLink used to connect the new Child process into the ring
(b) the 'identity' of the process instance to be created, namely a creation-order number; the recursion terminates when the value of Identity matches the value of Players supplied in the specification heading.

When the value of Identity is not equal to Players, a Child process is instantiated and connected to its neighbours by setting its ParcelIn and ParcelOut gates appropriately. When the value of Identity is equal to Players a Child process is instantiated with its connecting gates defined to complete the circle.

The description is now complete and shown in full in Appendix III. A number of observations can be made at this point:

1. Modelling the parcel has been worth while, as the description involved is abstract and yet serves as a good basis for an implementation.
2. Modelling children individually has not been easy and seems to yield little benefit. In particular, the LOTOS description of individual behaviour is much larger, more complex and considerably more difficult to understand than the equivalent natural-language text. It is easy to imagine how the formal description might be improved by having LOTOS features that allow processes to be instantiated and linked in a different way, but the description is still likely to be relatively complex.

3. The object-based description is clearly much larger than that of the event-based description, but mostly this is because of the introduction of definitions for the parcel and for each child who handles it. In fact, the two approaches applied at the same level of detail would tend to yield definitions of comparable size.
4. LOTOS can be used effectively for the description of data and the ordering of events. However, in its present form it cannot easily be used to express basic timing constraints such as a limitation that a child should not hold a parcel for more than a few seconds while music is playing, that a child should take only a few seconds to remove a layer of paper or that the music should stop at 'reasonable' intervals.

Overall, the object-based approach to formal description tends to yield a definition that is easier to understand than that produced by the event-based approach because of the direct link between the structure of the system concerned and that of its description. Event-based specifications have no such link, but they can be used effectively, for example, to define an initial statement of requirements.

## FROM FORMAL DESCRIPTION TO PROGRAM REPRESENTATION

A formal description of a system can serve as a precise statement of requirements against which to build an implementation. In some circumstances it may also be possible to use a formal description as a basis of software design. This is especially true of the object-based approach to system description, as illustrated for LOTOS, as it is similar to object-oriented design. It is thus possible to convert such descriptions into one of a range of modern programming notations in a largely mechanical way. In particular, there is a recognized correspondence between LOTOS data types and the Ada package , [32] and both languages have a concurrency model based on synchronized process (task) communication. For example, an Ada package corresponding to the LOTOS ParcelType developed earlier might be expressed as follows:

```
package ParcelType is
   type Parcel is limited private;
   subtype ParcelRange is Integer range 0. .Integer Last;
   function NewParcel (Size: ParcelRange) return Parcel;
   function WrappersRemain (P: Parcel) return Boolean;
   function UnWrap (P: Parcel) return Parcel;
private
   Parcel is ParcelRange;
  – | eqns (* equations *)
  – | for all n:  Nat
  – |    ofsort Bool
  – |       WrappersRemain  (NewParcel(0))        = false;
  – |       WrappersRemain (( NewParcel(succ( n)))= true;
  – | ofsort  Parcel
  – |       Unwrap  (NewParcel(0))                = NewParcel (0);
  – |       Unwrap  (NewParcel(succ(  n)))        = NewParcel (n);
end ParcelType;
```

The main points to note here are:

1. The operations in the LOTOS ParcelType definition have been realized by Ada functions; in general it may be necessary to use procedures in cases where error values need to be returned .[32]
2. The sort of the ParcelType is defined as an Ada *limited private type,* which imposes the required restrictions on the use of variables of this type, namely that they can only be manipulated within the ParcelType package.
3. In Ada the representation of the parcel must be defined before implementation—albeit in a private section; the representation used here is simply the positive integer subrange.
4. The equations of the LOTOS data-type definition serve as documentation for the ParcelType package and provide information on which to base an implementation.
5. The equations have a special introductory character ‘ | which allows a compiler to recognize the equations and possibly add implicit assertion code for the defined functions.

Further details, and many more examples of the use of algebraic data types definitions as specifications for Ada packages, may be found in Reference 32.

Event gates roughly correspond to Ada entries. Thus, for example, an Ada task implementing the Music process might have a definition part of the form:

```
task Music is
   entry MusicStarts;
   entry MusicStopped;
   entry PresentFound;
end Music;
```

In Ada, the communication of data values is handled by defining the names and types of the values concerned in a formal-parameter list for each entry affected. LOTOS event gates are more flexible in that they permit the number and type of the communicated values to be defined implicitly in an event offer.

The LOTOS choice expression roughly corresponds to the select statement in Ada. For example, the body of a Music task implementing the LOTOS Music process might be expressed as follows:

```
task body Music is
   Finished: Boolean : = False;
begin
   accept MusicStarts;
   while not Finished loop
      delay random time;
      accept MusicStops;
      select
         accept MusicStarts;
      or
         accept PresentFound do Finished : = True;
      end select;
   end loop;
end Music;
```

Note that the recursive LOTOS definition has been converted into a looping structure and that a random delay has been inserted (informally) to simulate the random playing intervals of the music.

One important difference between the LOTOS and Ada communication mechanisms is that in LOTOS the partners in an event have equal status, whereas in Ada the interaction is asymmetric—one task instigates an interaction, by making an entry call, and the receiving task accepts the interaction. A consequence of this asymmetry is that there are types of communication that can occur in LOTOS definitions but that cannot be expressed directly in Ada. The interaction between the Music process and each Child process is one example. As the MusicStops event is offered passively by the Music process a Child process is obliged to make an entry call but this then prevents the Child process from dealing with the arrival of the parcel. In general, there is no systematic method of dealing with structural clashes of this type and each must be considered separately as part of the refinement process. For LOTOS it has been suggested that each description be translated into an intermediate restricted form of the language, which would permit automatic translation to a target programming form. [33,34]

Further discussion of the relationship between LOTOS and Ada may be found in Reference 35.

## VERIFICATION

The preceding sections have discussed four different representations for the rules of the childrens' game of pass-the-parcel:

1. a natural-language description
2. a LOTOS description of part of the game expressed in an event-based style
3. a more detailed LOTOS description of the game expressed in an object-based style
4. an outline of a simulation model for the game expressed in Ada.

These descriptions are intended to be 'consistent' with each other in that no one of them contradicts any other, although some will contain greater detail than others. Such consistency can be verified informally by the systematic comparison of the descriptions using an inspection technique. [36] That is the only choice when comparing the natural-language description with any of the others. At present, it is also the only option when comparing LOTOS descriptions with Ada although, as discussed, there is scope for automatically translating some aspects of LOTOS into Ada, thereby limiting the amount of verification required.

The comparison of two LOTOS descriptions for consistency or *equivalence* can be done formally, but automated assistance is required for descriptions of any practical size, and the whole area is still very much the subject of research. The meaning of a LOTOS description is captured by its derivation tree showing all possible behaviors. Thus verifying the equivalence of two LOTOS descriptions effectively means comparing their derivation trees. Formally, a LOTOS description is a *labelled transition system* (see Reference 37 and Appendix I ), and verification means comparing two such systems according to an appropriate definition of equivalence.

'Various types of equivalence have been proposed. [6] The strongest is where descriptions are required to have identical derivation trees. In practice, however, a much

weaker notion of equivalence, known as *observational equivalence,* is more commonly used. Two descriptions are considered observationally equivalent if they cannot be distinguished by *external* observation. This means that when comparing one description with a refinement that introduces new events, the new events can be ignored as long as they do not affect externally-observed behaviour.

Note that the two LOTOS descriptions developed in this paper are not equivalent in any recognized sense because:

1. The object-based model additionally allows for the possibility that the game will not start if the parcel has no wrappers or there are not at least two children available to play the game.
2. The single parcel passed event in the event-based model is expanded as multiple events, each linking a distinct pair of communicating children.

A useful validation tool would therefore have to work round such differences, probably under human guidance.

Some verification tools have been developed for LOTOS but these have tended to be restrictive in various ways. Most commonly, the size and type of specifications is limited because of the memory needed to hold LOTOS derivations .[37,38] This particular problem, however, has recently been alleviated through an 'on the fly' technique for verifying that avoids the need to construct full LOTOS derivations initially.[39] Such tools are probably not adequate for routine use but they do demonstrate what can be achieved, in principle.

There is also an additional (long-term) possibility. This is based around the *interface equation* concept.[40] The idea here is that, in certain circumstances, it may be possible to generate, automatically, a definition of the interface (difference) between two given formal descriptions. Currently this theory has been developed for CCS, but it could be applied to LOTOS since the semantic content of the process algebras is similar. An interface equation is expressed in the form

$$(p|X)\backslash A \approx q$$

where $p$ and $q$ are the given definitions and $X$ is the definition to be derived. The '|' represents parallel composition and the '$\backslash A$' means that the set of actions, $A$, *is* internal to $p$ and $X$. (This corresponds to the LOTOS hide operator. ) The '$\approx$' denotes observational equivalence. Given certain restrictions, principally that the descriptions can be represented by finite-state machines, it is possible to derive $X$ automatically. Thus, if $p$ represents an object-based description and $q$ an event-based description, $X$ defines how one is linked to the other. This work is in the realms of long-term research but nevertheless holds some promise of additional automation in the verification process.

## CONCLUSION

This paper has illustrated how concurrent systems might be described using a process-oriented formal description language and discussed how such systems might be developed. Most emphasis has been placed on system description because it is believed that that is where the greatest return for effort is to be obtained. The act of constructing a formal model increases the developer's understanding of a system

and will often lead to the detection of errors masked in a natural-language description. These benefits justify the use of a formal model even if it is then largely ignored in subsequent development. A high-level model, focusing on the most significant aspects of a system, can be constructed with relatively little effort and provide a good return for that effort. More detailed models are justifiable if they help with system design and so form an integral step towards an implementation.

Languages such as LOTOS can be used to model software, the users' interaction with that software and even the process by which the software is produced. In principle, such notations can support a 'pencil and paper' specification technique. In practice, however, it seems likely that the provision of tools for the construction and analysis of specifications will greatly enhance the acceptability of a formal approach to description. Tools go hand-in-hand with methodology and so should improve as a deeper understanding of the role of formal languages in system development becomes clearer. The present paper has attempted to take a step towards that goal.

ACKNOWLEDGEMENTS

This paper has benefited from discussions with David Freestone, Hossein Rafsanjani, Rob Neely and Alan Stewart. We are very grateful for their comments and suggestions, and also for those of the referees. Parts of the paper was prepared by the first-named author during sabbatical visits to the British Telecom Research Laboratories at Martlesham Heath, Ipswich and to the Software Engineering Institute, Carnegie Mellon University, Pittsburgh. The paper is a contribution to the SCAFFOLD project, being undertaken collaboratively with York University and British Aerospace, and funded by Grant GR/G 03700 of the U.K. Science and Engineering Research Council.

## APPENDIX I: SUMMARY OF MAIN LOTOS FEATURES

The LOTOS language has two components:

(a) A *process algebra* based mainly on ideas used in CCS [12] and CSP. [13] This is used to express the temporal behaviour of a system.

(b) An *abstract data-type* component based on the algebraic language ACT ONE. [14] This is used to specify the data within a system in terms of their types or *sorts* and the *operations* to construct and manipulate them.

In general, a system is described in LOTOS as a hierarchy of nested process and type definitions.

The behaviour of a process is described by a *behaviour expression.* This is a combination of atomic *events* (or *actions)* and the instantiation of processes linked using operators provided by the language. Processes interact by sharing events, which may involve the interchange of data. The events through which a process can interact are declared as formal parameters in its definition—when a process is instantiated, corresponding actual parameters are given. In a similar way data values can be passed to processes through parameters. A process can be instantiated recursively to specify repeated behaviour.

Two basic processes are built into LOTOS: stop and exit. These represent inactivity and successful termination respectively. A special event $\delta$ is offered implicitly by exit.

A special event i is used explicitly to represent an action that does not involve interaction with any other process. It is internal to the process in which it appears.

The meaning of LOTOS operators is defined formally within the ISO standard in terms of their operational semantics. These are expressed as axioms and inference rules based on a system of labelled transitions. Using these it is possible to derive two things:

1. A behaviour expression's *initials,* the set of possible actions in which it can immediately take part. These actions are offered to the expression's environment for interaction. They can be defined using a function with the following signature:

   initials: behaviour expression $\rightarrow$ set of events

2. The expression specifying the *behaviour* subsequent to the performance of one of these initials. For an action to occur it must be accepted by a matching offer in the environment. The effect of this is defined using *axioms* in the simple cases of exit and *action prefix* expressions, plus *inference rules* to derive results for more complicated behaviour expressions.

The semantics of each operator used in basic LOTOS are given below. In each case they are first described informally. The formal axioms and inference rules defining the effect of each operator within a behaviour expression are then given, followed by the definition of the *initials* function derived from them. In the discussion the following symbols are used:

(a) The set operators for inclusion, exclusion, union and intersection ( $\in$, $\notin$, $\cup$, $\cap$).
(b) $\varnothing$, the empty set.
(c) *B, B* 1, *B* 2 are behaviour expressions.
(d) $g \cup G$ where $G$ is the set of user-defined actions.
(e) i represents the unobservable internal action.
(f) $\mu \in$ Act where Act = $G \cup \{ i \}$, i.e. the set of explicit actions.
(g) $S = [ g_1 .. g_n ]$, a finite sequence of user-defined action-names.
(h) $\delta$ represents successful termination.
(i) $g^+ \in G^+$ where $G^+ = G \cup \{ \delta \}$, i.e. the set of observable actions.
(j) $\mu^+ \in Act^+$ where $Act^+ = Act \cup \{ \delta \}$, i.e. the set of all actions.
(k) $g/g'$ represents the replacement of occurrences of the name $g$ by $g'$.
(1) $\Phi = [ g_1/g'_1 .. g_n/g'_n ]$ is a sequence of such replacements.

**Inactivity** ( stop )

Stop defines a totally inactive process that cannot engage in any events. Therefore there are no appropriate axioms or inference rules and the initials function returns the empty set:

initials ( **stop** ) = $\varnothing$

**Successful termination** ( exit )

Exit represents successful process termination. It is defined as the offering of the special event δ. If this offer is accepted by the environment the process becomes inactive, equivalent to stop:

$$\text{exit} \text{ — } \delta \rightarrow \text{stop}$$

This axiom can be read as 'the process exit may perform the event δ and transform into the process stop'. The initials of exit is the singleton set containing δ :

$$\text{initials (exit)} = \{ \delta \}$$

**Action prefix (;)**

Any behaviour expression can be prefixed by an action. For example μ; *B* means that action μ is followed by (or prefixes) behaviour *B*. Action prefix is the basic building-block from which sequences of actions can be composed into processes:

$$\mu; B \text{—} p \rightarrow B$$
$$\text{initials } ( \mu ; B ) = \mu$$

**Choice ([])**

*B* 1 [] *B* 2 means that either the behaviour *B* 1 or *B* 2 can occur. The outcome depends on the events offered by the environment unless the initial events of *B* 1 and/or *B* 2 are identical or involve the internal event i. In this case the choice is non-deterministic between *B* 1 and *B* 2. Each choice can be guarded by a predicate; only those events whose predicates evaluate to true are allowed to occur. The inference rules for choice expressions state that if the initial action of either subexpression occurs to produce a resulting behaviour expression (as shown above the horizontal line). then the overall construct will perform the same action to produce the same resulting expression (shown below the line):

$$\frac{B1 \text{—} \mu^{+} \rightarrow B1'}{B1 [] B2 \text{—} \mu^{+} \rightarrow B1'}$$

$$\frac{B2 \text{—} \mu^{+} \rightarrow B2'}{B1 [] B2 \text{—} \mu^{+} \rightarrow B2'}$$

The initials of a choice expression is the union of the initials of the individual subexpressions:

$$\text{initials } ( B1 [] B2) = \text{initials } ( B1) \cup \text{initials } ( B2)$$

**Parallel composition** (|[ *a,b* ,...]|)

*B* 1 | [ *a,b,...* ] | *B* 2 means that *B* 1 or *B* 2 occur in parallel and share, or synchronize on, the events listed within the brackets. Two special cases of this operator have

special symbols. Where *no* events are shared by the processes the events from each are interleaved. This can be represented as $B1 \,|||\, B2$. Where *all* events are shared, the behaviour is represented by $B1 \,||\, B2$. The implicit event in successful termination $\delta$ is always shared between parallel processes. This means that behaviors composed in parallel always terminate together.

There are three inference rules for parallel composition. The first two express the effect of events that are not shared between processes (i.e. $\mu \notin S$); the third those that are ($g^+ \in S \cup \{\delta\}$):

$$\frac{B1 \xrightarrow{\mu} B1',\ M \notin S}{B1 \,|S|\, B2 \xrightarrow{\mu} B1' \,|S|\, B2}$$

$$\frac{B2 \xrightarrow{\mu} B2',\ \mu \notin S}{B1 \,|S|\, B2 \xrightarrow{\mu} B1 \,|S|\, B2'}$$

$$\frac{B1 \xrightarrow{g^+} B1',\ B2 \xrightarrow{g^+} B2',\ g^+ \in S \cup \{\delta\}}{B1 \,|S|\, B2 \xrightarrow{g^+} B1' \,|S|\, B2'}$$

$$\text{initials}(B1 \,|S|\, B2) = (\text{initials}(B1) - S) \cup (\text{initials}(B2) - S) \cup (\text{initials}(B1) \cap \text{initials}(B2) \cap S)$$

**Disabling ([>)**

$B1 \,[>\, B2$ means that the behaviour $B1$ will be interrupted and not resumed if an event occurs in $B2$. If $B1$ terminates naturally before $B2$ interrupts then the events in $B2$ never occur. This is expressed in three inference rules: for the occurrence of an event in $B1$, for the termination of $B1$ and for the occurrence of an event in $B2$. The initials of a disable expression are the union of its two parts:

$$\frac{B1 \xrightarrow{\mu} B1'}{B1 \,[>\, B2 \xrightarrow{\mu} B1' \,[>\, B2}$$

$$\frac{B1 \xrightarrow{\delta} B1'}{B1 \,[>\, B2 \xrightarrow{\delta} B1'}$$

$$\frac{B2 \xrightarrow{\mu^+} B2'}{B1 \,[>\, B2 \xrightarrow{\mu^+} B2'}$$

$$\text{initials}(B1 \,[>\, B2) = \text{initials}(B1) \cup \text{initials}(B2)$$

**Sequential composition (>>)**

$B1 >> B2$ signifies that when $B1$ successfully terminates (represented by the special event $\delta$) the behaviour $B2$ is enabled. The $\delta$ event that triggers $B2$ is not visible to the environment and so is equivalent to an internal event i. Inference rules are needed for the effect of a normal event in $B1$ and that of $B1$'s termination. The initials of the compound expression are simply those of the enabling process:

$$\frac{B1 \xrightarrow{\mu} B1'}{B1 >> B2 \xrightarrow{\mu} B1' >> B2}$$

$$\frac{B1 \xrightarrow{\delta} B1'}{B1 >> B2 \xrightarrow{i} B2}$$

initials ( $B1 >> B2$) = initials ( $B1$)

**Hiding** ( hide . . . in )

*Hiding* makes named events internal to a behaviour expression and thus unavailable for interaction with the environment, essentially giving them the characteristics of the internal event i. Inference rules are stated as follows:

$$\frac{B \xrightarrow{g} B',\ g \in \{g_1 \ldots g_n\}}{\text{hide } g_1 \ldots g_n \text{ in } B \xrightarrow{i} B'}$$

$$\frac{B \xrightarrow{\alpha^+} B',\ \mu \{g_1 \ldots g_n\}}{\text{hide } g_1 \ldots g_n \text{ in } B \xrightarrow{\mu^+} B'}$$

initials ( hide $g_1 \ldots g_n$ in $B$ ) = initials ( $B$ ) i /gn/.. .i /$g_n$

**Process instantiation**

Process instantiation is the main structuring tool within the behaviour part of a LOTOS specification. It is used to decompose complex constructs into simpler and more manageable units. Its use also allows the parametrization of behaviour expressions and, by recursive instantiation, the specification of repetitive behaviour. The effect of instantiating a process is that of substituting the instantiation by the behaviour expression given in the process's definition. All occurrences of events given as formal parameters are replaced by the corresponding actual parameters. Given a process definition

process $P [g'_1 \ldots g'_n] := B_p$ endproc

process instantiation is expressed by the inference rule

$$\frac{B_p[g_1/g'_1 \ldots g_n/g'_n] \xrightarrow{\mu^+} B'}{P[g_1 \ldots g_n] \xrightarrow{\mu^+} B'}$$

The effect of the renaming $[g_1/g'_1 \ldots g_n/g'_n]$ on a behaviour expression is given by two inference rules:

$$\frac{B \xrightarrow{g'} B',\ g/g' \in \phi}{B\phi \xrightarrow{g} B'\phi}$$

$$\frac{B \xrightarrow{\mu^+} B',\ u^+ \notin \{g'_1 \ldots g'_n\}}{B\phi \xrightarrow{\mu^+} B'\phi}$$

The initials of a process instantiation are those of the behaviour expression in the corresponding process definition renamed in line with the formal and actual parameters:

initials ( $P [g_1 \ldots g_n]$) = initials ( $B_p$)$[g_1/g'_1 \ldots g_n/g'_n]$

**Data types**

The data-type component of LOTOS is a standard algebraic notation. Each data type has a *name*, a *sort name* used to identify the values of the type, a set of operations (introduced by the keyword opns ), defining the domain and range of the operations defined within the type, and a set of equations (introduced by the keyword eqns ), defining properties of the operations. In simple terms, the **opns** define the syntax of the operations and the eqns define the semantics of the operations.

LOTOS provides no built-in data types—all data and operations must be defined in the language. However LOTOS does provide for a library of types, and the ISO definition for LOTOS includes a standard set of basic types including, for example, natural number and Boolean.

## APPENDIX II: PASS-THE-PARCEL SPECIFICATION IN CONSTRAINT-ORIENTED STYLE

```
specification PassTheParcel [ParcelSupplied, ParcelPassed, MusicStarts,
MusicStops, LayerRemoved, PresentFound, CallToTea]: exit
  behaviour
    (InitialMusic [MusicStarts] >>
      PrepareParcel [ParcelSupplied] >>
        Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved,
        PresentFound])
    [> Tea [CallToTea]
    where

    process InitialMusic [MusicStarts]: exit :=
      MusicStarts; exit
    endproc (* InitialMusic *)

    process Prepare Parcel [ParcelSupplied]: exit : =
      ParcelSupplied; exit
    endproc (* PrepareParcel *)

    process Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved,
    PresentFound]: exit :=
      ( ParcelPassed;
        Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved,
        PresentFound])
        []
      ( MusicStops;
        LayerRemoved;
        (( MusicStarts;
           Parcel Passed;
           Circulate [ParcelPassed, MusicStarts, MusicStops, LayerRemoved,
           PresentFound])
           []
        ( PresentFound; exit )))
    endproc (* Circulate *)

    process Tea [CallToTea]: exit :=
```

```
      CallToTea; exit
    endproc (* Tea *)
endspec (* PassTheParcel *)
```

## APPENDIX III: PASS-THE-PARCEL SPECIFICATION IN OPERATIONAL-MODELLING STYLE

```
specification PassTheParcel (ParcelSize: Nat, Players: Nat): exit
  library NaturalNumber, Boolean endlib

  behaviour
    [(Players le succ(0)) or (ParcelSize eq 0)]-> ( exit )
    []
    [Players gt succ(0) and (Parcel Size gt 0)]-> (Game (ParcelSize, Players) [>
    Mother)
    where

    process Mother: exit : =
      i ; exit
    endproc (* Mother *)

    process Game (ParcelSize: Nat, Players: Nat): exit :=
      hide ParcelSupplied, MusicStarts, MusicStops, PresentFound in
      (Father [MusicStarts, ParcelSupplied] (ParcelSize)
        |[ParcelSupplied]|
      Children [ParcelSupplied, MusicStarts, MusicStops, PresentFound]
      (Players))
        |[MusicStarts, MusicStops, PresentFound]|
      Music [MusicStarts, MusicStops, PresentFound]
      where

      type ParcelType is NaturalNumber, Boolean
        sorts Parcel
        opns (* operations *)
          New Parcel:        Nat      –> Parcel
          Wrappers Remain:   Parcel   –> Bool
          Unwrap:            Parcel   –> Parcel
        eqns (* equations *)
          forall n: Nat
          ofsort Bool
            WrappersRemain (NewParcel (0))       = false;
            WrappersRemain (NewParcel(succ( n))) = true;
          ofsort Parcel
            Unwrap (NewParcel(0))       = NewParcel (0);
            Unwrap (NewParcel(succ( n))) = NewParcel (n);
        endtype (* ParcelType *)

        process Father [MusicStarts, ParcelSupplied] (ParcelSize: Nat): exit : =
          MusicStarts; ParcelSupplied ! NewParcel (ParcelSize); exit
        endproc (* Father *)
        process Music [MusicStarts, MusicStops, PresentFound]: exit : =
```

```
   MusicStarts;
   MusicStops;
   ((Music [MusicStarts, MusicStops, PresentFound])
   []
   (PresentFound; exit ))
endproc (* Music *)

process Children [ParcelSupplied, MusicStarts, MusicStops, pre-
sentFound] (Players: Nat): exit :=
   hide FirstLink, NextLink in
   (Child [ParcelSupplied, FirstLink, NextLink, MusicStarts, MusicStops,
   PresentFound] (true)
   |[FirstLink, NextLink, PresentFound]|
   StartChild [ParcelSupplied, FirstLink, NextLink, MusicStarts,
   MusicStops, PresentFound] (succ (succ(0)), players))
   where
   process Child [ParcelSupplied, Parcelin, ParcelOut, MusicStarts,
   MusicStops, PresentFound] (ParcelFromFather: Bool): exit : =
      [ParcelFromFather] –> (* first child accepts parcel from Father *)
         (ParcelSupplied ? TheParcel: Parcel;
         ParcelAction [ParcelSupplied, Parcelin, ParcelOut, MusicStarts,
         MusicStops, PresentFound] (Theparcel))
      []
      [not (ParcelFromFather)] –>
         (( PresentFound; exit)
         []
         (Parcel ? TheParcel: Parcel;
         ParcelAction [ParcelSupplied, Parcelin, ParcelOut, MusicStarts,
         MusicStops, PresentFound] (TheParcel)))
      where
      process ParcelAction [ParcelSupplied, Parcelin, ParcelOut,
      MusicStarts, MusicStops, PresentFound] (TheParcel: Parcel):
      exit : =
         ( ParcelOut ! TheParcel;
            Child [ParcelSupplied, Parcelin, parcelout, MusicStarts,
            MusicStops, PresentFound] (false))
         []
         ( MusicStops;
            i; (* remove layer of paper *)
            ( let UnwrappedParcel: Parcel = Unwrap (TheParcel) in
               ( [Not (Wrappers Remain (Unwrapped Parcel))] ->
               (PresentFound; exit )
               []
               [WrappersRemain (UnwrappedParcel) |–>
                  ( MusicStarts; ParcelOut ! UnwrappedParcel;
                    Child [ParcelSupplied, Parcelin, ParcelOut, MusicStarts,
                    MusicStops, PresentFound] (false))
            ) (* selection on WrappersRemain or not WrappersRemain *)
```

```
            ) (* scope of UnwrappedParcel *)
          ) (* MusicStops selection *)
        endproc (* ParcelAction *)
      endproc (* Child *)

      process StartChild [ParcelSupplied, FirstLink, PreviousLink, MusicStarts,
      MusicStops, PresentFound] (Identity: Nat, Players: Nat): exit :=
        [Identity eq Players] -> (* last child *)
          (Child [ParcelSupplied, PreviousLink, FirstLink, MusicStarts,
           MusicStops, PresentFound] (false))
          []
        [Identity lt Players] –> (* child other than the first or last instantiated
        *)
          ( hide NextLink in
            Child [ParcelSupplied, PreviousLink, NextLink, MusicStarts,
            MusicStops, PresentFound] (false)
            |[NextLink, PresentFound]|
            StartChild [ParcelSupplied, FirstLink, NextLink, MusicStarts,
            MusicStops, PresentFound] (succ (Identity), Players))
    endproc (* StartChild *)
  endproc (* Children *)
endproc (* Game *)
endspec (* PassTheParcel *)
```

## REFERENCES


1. J. Woodcock and M. Loomes, *Software Engineering Mathematics,* Pitman, 1988.
2. D. C. Ince, *An Introduction to Discrete Mathematics and Formal System Specification,* Clarendon Press, 1988.
3. C. B. Jones, *Systematic Software Developing Using VDM,* Prentice Hall, 1986.
4. M. Nielsen, K. Havelund, K. R. Wagner and C. George, 'The RAISE language, methods and tools', *Formal Aspects of Computing,* **1,** 85–114 (1989).
5. E. Brinksma (ed.), *Information Processing Systems—Open Systems Interconnection—LOTOS—A Formal Technique Based on the Temporal Ordering of Observational Behaviour,* ISO IS 8807, 1988.
6. T. Bolognesi and E. Brinksma, 'Introduction to the ISO Specification Language LOTOS', in P. H. J. van Eijk, C. A. Vissers and M. Diaz (eds), *The Formal Description Technique LOTOS,* North Holland, 1988.
7. D. W. Bustard, J. W. G. Elder and J. Welsh, *Concurrent Program Structures,* Prentice Hall, 1988.
8. I. Hayes (ed.), *Specification Case Studies,* Prentice-Hall, 1987.
9. J. A. Gougen and J. J. Tardo, 'An Introduction to OBJ: a language for writing and testing formal algebraic program specifications', in N. Gehani and A: D. McGettrick (eds), *Software Specification Techniques,* Addison-Wesley, 1986.
10. J. V. Guttag, J. J. Horning and J. Wing, 'Larch in five easy pieces', *Report 5,* Digital Systems Research Centre, 1985.
11. R. M. Burstall and J. A. Gougen, 'An introduction to specification using CLEAR', in N. Gehani and A. D. McGettrick (eds), *Software Specification Techniques,* Addison-Wesley, 1986.
12. R. Milner, *A Calculus of Communicating Systems,* Lecture Notes in Computer Science, Springer-Verlag, 1980.
13. C. A. R. Hoare, *Communicating Sequential Processes,* Prentice Hall, 1985.
14. H. Erhig and B. Mahr, *Fundamentals of Algebraic Specification 1,* Springer-Verlag, Berlin, 1985.

15. V. Carchiolo, A. Fare, O. Mirabella, G. Pappalardo and G. Scollo, 'A LOTOS specification of the PROWAY highway service', *IEEE Trans. Computers,* **C-35,** (11), 949–968 (1986).
16. E. Brinksma, 'Experience with and future of LOTOS as a specification language', in R. Saracco and P. A. J. Tilanus (eds), *SDL '87: State of the Art and Future Trends,* North Holland, 1987.
17. K. J. Turner (ed.), *FORTE '88, Proc. 1st International Conference on Formal Description Techniques,* North Holland, 1988.
18. P. H. J. van Eijk, C. A. Vissers and M. Diaz, *The Formal Description Technique LOTOS,* Elsevier, 1989.
19. P. H. J. van Eijk, 'Tools for LOTOS style transformation', in S. Vuong (ed.), *Formal Description Techniques '89,* North Holland, 1989.
20. C. Vissers, G. Scollo and M. van Sinderen, 'Architectural and specification style in formal descriptions of distributed systems', *Protocol Specification, Testing and Verification VIII, 1988,* pp. 189–204.
21. M. Vigder and R. J. A. Buhr, 'Using LOTOS in a design environment', in K. Parker and G. Rose (eds), *Formal Description Techniques IV,* North-Holland, 1992.
22. W. Reisig, *Petri Net;: an Introduction,* Springer Verlag, 1985.
23. M. W. Shields, *Finite State Autotomata,* Blackwell, 1989.
24. ISO, *Proposed Draft Addendum to ISO 8807:1988 on G-LOTOS,* ISO/IEC JTC1/SC21, 1990.
25. K. Naik and B. Sarikya, 'Testing Communication Protocols', *IEEE Software,* January 1992, pp. 27–37.
26. D. W. Bustard, M. T. Norris and R. A. Orr, 'A pictorial approach to the animation of process-oriented formal specifications', *IEE Software Engineering Journal,* **3,** (4), 114–118 (1988).
27. L. Logrippo, A. Obaid, J. P. Briand and M. C. Fehri, 'An interpreter for LOTOS, a specification language for distributed systems', *Software–Practice and Experience,* **18,** (4), 365–385 (1988).
28. A. Winstanley and D. W. Bustard, 'EXPOSE: an animation tool for process-oriented specifications', *Software Engineering Journal,* **6,** (6), 463–475 (1991).
29. P. H. J. van Eijk, 'The LOTOSPHERE integrated tool environment LITE', in K. J. Turner (ed.), *Formal Description Techniques IV,* North Holland, 1992.
30. W. C. Hetzel, *The Complete Guide to Software Testing,* 2nd edn, QED Information Sciences Inc., 1988.
31. R. Milner, *Communication and Concurrency,* Prentice Hall, 1989.
32. I. Somerville and R. Morrison, *Software Development with Ada,* Addison-Wesley, 1985.
33. P. van Eijk, H. Kremer and M. van Sinderen, 'On the use of specification styles for automated protocol implementation from LOTOS to C, in L. Logrippo, R. L. Probert and H. Ural (eds), *Protocol Specification, Testing and Verification X,* North Holland, 1990, pp. 157–168.
34. K. J. Turner, 'A LOTOS-based development strategy', in S. Vuong (ed.), *Formal Description Techniques '89,* North Holland, 1989.
35. D. W. Bustard, M. T. Norris and R. A. Orr, 'Formalizing the design of Ada systems using LOTOS', *Proc. Ada Europe,* Madrid, June 1989.
36. T. Gilb, *Principles of Software Engineering Management,* Addison-Wesley, 1989.
37. T. Bolognesi and M. Caneve, 'SQUIGGLES: a tool for the analysis of LOTOS specifications', in K. J. Turner (ed.), *Formal Description Techniques' 88,* North-Holland, 1988.
38. S. S. Aujla and M. Fletcher, 'The Boyer–Moore theorem prover and LOTOS', in K. J. Turner (ed.), *Formal Description Techniques '88,* North-Holland, 1988.
39. J-C. Fernandez and L. Mounier, 'Verifying bisimulations on the fly', *Proc. Formal Description Techniques '90,* Madrid, 1990.
40. G. Martin, M. T. Norris and M. W. Sheilds, 'The interface equation', *Mathematics Bulletin,* **25,** (6), 15–16 (1989).