EXPOSE: an animation tool for process-oriented specifications

by Adam C. Winstanley and David W. Bustard

This paper describes and evaluates EXPOSE, an animation tool for process-oriented formal specifications of concurrent systems. EXPOSE takes as input the text of a formal specification and, from it, generates static views (interpretations) of the specification structure and dynamic views of the behaviour of the system specified. The views are constructed and explored using the hypermedia facilities of the Apple Macintosh HyperCard system. EXPOSE has been implemented experimentally for LOTOS, but most of the animation concepts and techniques described are relevant to other process-oriented languages, such as CCS and CSP.

1 Introduction

A formal notation provides an alternative to natural language when defining aspects of a computing system. It can either replace a natural language description or help make its interpretation more exact. The process of producing a formal specification gives the developer an improved understanding of the system described and yields a precise definition of requirements against which an implementation can be verified. Unfortunately, despite these advantages, the use of formal specification in system development is still the exception, rather than the rule. One reason for the lack of uptake is that both developers and customers find such specifications difficult to understand. The two main contributing factors are

- the mathematical notations involved are symbol-based, rather than text-based, and so must be learned before reading fluency can be attained.
- clear communication tends to rely on reinforcing concepts through repetition, perhaps by presenting the concepts in several different ways, or by using illustrations; formal specifications are concise descriptions that avoid redundancy, and so their meaning is not always immediately apparent.

In practice, the comprehension of formal specifications can be a problem with relatively short descriptions of only one or two pages, and for large specifications of 50 pages or more, for example, the difficulties encountered may be severe.

A previously published paper [1] discussed how an understanding of process-oriented specifications, in particular, can be improved by using *animation* techniques. These involve the generation of alternative interpretations or *views* of a formal description that help to illuminate its meaning; static views show the structure of a specification, whereas dynamic views reveal the behaviour of the system described. Subsequent papers identified possible static views for process-oriented specifications [2] and described an experimental specification browser [3]. EXPOSE (EXperimental Process-Oriented Specification Elucidator) is an animation tool that builds on this earlier work. It operates by taking the text of a complete process-oriented description as input and, from it, automatically generating various static and dynamic views of the description that may be examined using hypermedia techniques.

The basic goals guiding the development were

☐ *to focus on the needs of the specification developer, rather than those of the customer.* The views produced relate primarily to the nature of the specification, rather than to the system being specified. However, as the two are closely related, EXPOSE does offer some improvement in customer presentation.

☐ *to support existing specification developers who are familiar with the textual notation involved.* EXPOSE has been designed to assist those who are currently developing specifications in a textual form. It is expected, however, that the improvements made to the underlying development process will also help those new to this area.

☐ *to use a combination of graphical and textual representations in specification views.* In effect, views of a specification focus on relationships among its components. Some of these relationships can be expressed in a graphical form, some in a textual form and some in either representation. Graphical representations are given prominence, but, where

appropriate, both styles are made available to cater for the different preferences of users.

□ *to keep views simple.* Each view covers a particular inter-relationship among specification components, and, where necessary, descriptive information is suppressed to enable that interrelationship to be evident; in most cases, the suppressed information can be obtained when required by selecting a representative icon in the view.

EXPOSE has been implemented experimentally for the formal description language LOTOS [4], using the hypermedia facilities of the Apple Macintosh HyperCard system [5, 6]. In this paper, we provide some background to the LOTOS notation and an example of its use; we present details of the static and dynamic views of LOTOS specifications that are supported by EXPOSE; and we discuss the implementation of EXPOSE, and assess its strengths and weaknesses.

2 The LOTOS language

Within the broad range of formal notations, there are several aimed at the specification, design and analysis of concurrent systems, including CCS (Calculus of Communicating Systems) [8], CSP (Communicating Sequential Processes) [9], LOTOS (Language Of Temporal Ordering Specifications) [4] and ACP (Algebra of Communicating Processes) [10]. This paper is mainly concerned with LOTOS, but many of the concepts discussed are equally applicable to the other notations in this category. These are all based on a model of interacting sequential processes and have many basic features in common. The nature of these languages make them amenable to symbolic execution, and the definition of LOTOS, in particular, was formulated with this possibility in mind [11].

LOTOS is a process-oriented description language developed for the definition of OSI (Open Systems Interconnection) protocol standards. In practice, it is equally applicable to the definition of many types of concurrent system. Its model of concurrency is based on those of CCS and CSP, and it includes an algebraic data-typing facility largely drawn from ACT ONE [12]. A full international Standard for LOTOS was released by ISO in February 1989 [4]. Useful introductions and tutorial guides to the language can be found in References 7, 13 and 14; additional research material can be found in References 14–16. For a general discussion of the use of tools and development methods with LOTOS, see Reference 17. This paper also identifies the three Esprit projects, PANGLOSS, SEDOS and LOTOSPHERE, that have contributed most to the production of such tools and methods.

As an example of the use of LOTOS, consider how a formal description of the basic behaviour of a photocopying machine might be expressed. The machine has two buttons; one to request a copy to be made, and the other to switch off the machine. An LCD shows the message *out of paper* when a sensor on the paper feed-tray detects that there is no paper left on attempting to make a copy; once paper is loaded, the copy is produced.

A LOTOS description, known as a *specification*, consists typically of a hierarchy of process and data type definitions. Processes describe system behaviour, and data types describe the data manipulated within the system. Behaviour is defined by placing constraints on the order of system events. The set of events that are significant varies with the level of system *observation*. For example, a photocopier service engineer will be aware of events associated with the internal operation of the machine, whereas a user of the photocopier may only know about external events. The events of this latter category might be as follows:

event	**meaning**
copyrequest	the depression of the copy button
producecopy	the emergence of a copy
outofpaper	the display of the message that the paper tray is empty
paperloaded	the return of the paper tray to the machine after loading paper
poweroff	the depression of the button to switch off the machine

Such visible events would be named in the heading of a LOTOS specification of the photocopier:

specification *photocopier* [copyrequest, producecopy, outofpaper, paperloaded, poweroff] : **exit**

The heading also includes the name of the specification and an indication (**exit**) that the system described terminates. The events (strictly *event gates*) define the interaction (conceptually *synchronisation*) that can occur between the photocopier and its environment.

A LOTOS specification is a single process. Typically, the behaviour of that process is described in terms of local subprocesses, which, in turn, may be described by further subprocesses. For example, a top-level description of the photocopier might be represented by two processes; thus

```
behaviour
  normaloperation [copyrequest, producecopy,
                                outofpaper, paperloaded]
  [>
  (poweroff; exit)
```

The first process, *normaloperation*, describes the behaviour of the photocopier while it is available for copying. The second process (poweroff; **exit**) is implicit and unnamed. It describes the termination of the system when the power supply is switched off. Connecting the two processes is a *disable operator* [>. It indicates that the occurrence of an event in the implicit process, the *poweroff* event, will terminate or *disable* all activity described by the *normaloperation* process.

Normaloperation can be represented by two communicating processes; thus

```
process normaloperation [copyrequest, producecopy,
                         outofpaper, paperloaded] : noexit :=
  hide page in
      (copier [copyrequest, producecopy, page]
      |[page]|
      papertray [page, outofpaper, paperloaded])
endproc (* normaloperation *)
```

Process *copier* describes the sequence of events that lead to the production of a copy. Process *papertray* describes how pages are provided to the copier and also the procedure followed when the paper supply needs to be replenished.

These processes operate in parallel and have a shared event *page*, denoting the transfer of a page from the paper tray to the copier. This link is described by the parallel operator |[page]| between the two process references. The page event is local to the normal operation description (and internal to the photocopier), and so is *hidden* from the rest of the specification. Note that the *normaloperation* process has no explicit exit, as this occurs only as a consequence of the power being switched off.

The copier process can be described by a sequence of events; thus

```
process copier [copyrequest, producecopy,
                                    getpage] : noexit :=
  copyrequest;
  getpage;
  producecopy;
  copier [copyrequest, producecopy, getpage]
endproc (* copier *)
```

After a copy request is received, the copier obtains a page from the paper tray, produces a copy and returns to its original state, as represented by the recursive instantiation of the *copier* process.

The *papertray* process might have the following form:

```
process papertray [sendpage, outofpaper,
                                    paperloaded] : noexit :=
  (sendpage; papertray [sendpage, outofpaper,
                                    paperloaded])
  [ ]
  (outofpaper; paperloaded; papertray
                  [sendpage, outofpaper, paperloaded])
endproc (* papertray *)
```

The behaviour of the paper tray depends on the availability of paper. The two possibilities are defined in a choice expression, with the alternatives separated by the operator [ ]. If a page is present, it is sent to the copier and the paper tray returns to its initial state; otherwise, the out-of-paper event is reported and a return made to the initial state once paper has been loaded (or at least the paper tray taken out and returned).

The full LOTOS description of the photocopier is as follows:

```
specification photocopier [copyrequest, producecopy,
          outofpaper, paperloaded, poweroff] : exit
(*This is a specification of the behaviour of a
                        simple photocopying machine. *)
  behaviour
    normaloperation [copyrequest, producecopy,
                                outofpaper, paperloaded]
    [>
    (poweroff; exit)
  where
  process normaloperation [copyrequest, producecopy,
            outofpaper, paperloaded] : noexit :=
    hide page in
      (copier [copyrequest, producecopy, page]
      |[page]|
      papertray [page, outofpaper, paperloaded])
    where
  process copier [copyrequest, producecopy,
                                    getpage] : noexit :=
    copyrequest;
    getpage;
    producecopy;
    copier [copyrequest, producecopy, getpage]
  endproc (* copier *)
  process papertray [sendpage, outofpaper,
                                paperloaded] : noexit :=
    (sendpage; papertray [sendpage, outofpaper,
                                paperloaded])
    [ ]
    (outofpaper; paperloaded; papertray [sendpage,
                        outofpaper, paperloaded])
  endproc (* papertray *)
endproc (* normaloperation *)
endspec (* photocopier *)
```

This example has provided an informal introduction to the main features of the behavioural component of LOTOS, in order to provide a flavour of the process-oriented approach to formal description and to show the typical appearance of such a specification. In the rest of the paper, we consider how alternative views of process-oriented descriptions might be presented to help make their meaning clearer.

A fuller, more formal definition of the LOTOS features shown in the example above may be found in the Appendix. One notable aspect of LOTOS not illustrated by the photocopier example is its data type facility. For the photocopier, the only data items involved are the pages being copied, and these do not require explicit representation. In specifications where such representation is needed, abstract data types are used to define operations on the data and explicit data values are attached to events in which data are communicated. This area of LOTOS is not explored here. The purpose of this paper is not so much to discuss LOTOS, but rather show how the structure and implied behaviour of process-oriented descriptions, in general, might be presented. Below we discuss the use of HyperCard in providing specification views of this type.

3 Static views

EXPOSE presents static and dynamic views of LOTOS specifications using HyperCard [5, 6]. This is a hypermedia system currently supplied free with each new Apple Macintosh computer. HyperCard is used mainly to build simple databases and to construct application prototypes. The decision to use HyperCard for EXPOSE, rather than develop a more integrated self-contained system, was mainly based on a perceived need for flexibility. When EXPOSE was initially designed (in 1988), a graphical version of LOTOS [19] was under development through ISO, and it was felt desirable to retain compatibility with any proposals they produced. HyperCard was seen as a way to attain that flexibility, as it would enable changes of interface representation to be implemented relatively quickly. It did have several obvious technical disadvantages when first released, such as slow performance and restrictions on the type of drawings that could be produced, but it seemed likely that many of these problems would disappear in later releases. Improvements to Hypercard have indeed occurred but much slower than expected. Overall, however, the flexibility that has been

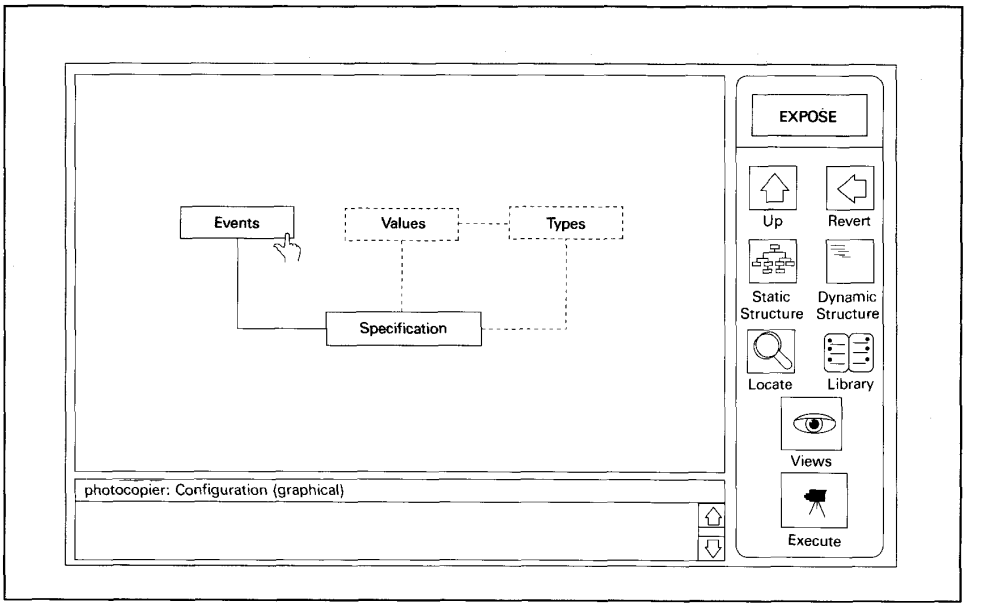

Fig. 1 Top view of photocopier specification

achieved with HyperCard still leaves it a valid choice for use in an experimental tool such as EXPOSE. Further, detailed comments on the limitations of HyperCard may be found in the concluding Section of this paper.

HyperCard supports the manipulation and presentation of textual and graphical data, and provides a means of communicating with other applications. Data are stored on *cards* (screens) that may be linked as desired in *stacks*. Data are held as either bit-mapped images or as text in *fields*. Movement from one card to another is usually achieved by the selection of *buttons*. HyperCard stacks can be constructed using operations issued at the keyboard. Alternatively, all such operations can be programmed explicitly in HyperTalk [18]. EXPOSE makes use of this latter facility to build animation stacks from LOTOS specifications.

EXPOSE constructs LOTOS specification views as a collection of HyperCard cards. For example, Fig. 1 shows the top-level card for the photocopier specification.

Each card is divided into three parts.

- The *main window* (top left) contains a *view*; the components of each view either have links to other views or yield further information when selected.
- The *elaboration window* (bottom left) is used to display the further information on selected components.
- The *control panel* (right) contains buttons that are used to browse through the views. (Note that the current implementation is based on HyperCard 1.2, which is limited to showing one card at a time on the screen. Thus, views are inspected individually.)

The view in Fig. 1 identifies possible environment links to the specification. In general, a specification may define external event interactions, values that instantiate the specification and data types that are global to the specification. A rectangular box is used to denote each of these specification components, and the connecting lines indicate relationships among the components. Components that are not applicable in any instance are shown as faded. Thus, in this case, the diagram reveals that the specification refers to environment events, and that there are no instantiating values or global types involved.

EXPOSE diagrams, in general, have been designed to clarify various aspects of a formal description. More specifically, the diagrams are intended to highlight the main static and dynamic relationships of interest in a specification.

The static views for LOTOS identify

☐ the specification *configuration*; the links between a specification and its environment (as illustrated in Fig. 1).
☐ the *library types* used in the specification.
☐ the *nesting* of process and type definitions.
☐ the *instantiation* interdependence of processes.
☐ the *types* defined in the specification.
☐ the permitted *behaviour* of each process, defined in terms of the interdependence of the events in which that process may participate.

These views are arranged at three logical levels, as shown in Fig. 2. At each level, there is a main graphical view and one or more alternative interpretations of that view. The interpretations are connected in a cyclic fashion and may be inspected successively, using the *Views* button in the control panel. For example, the top level shows three views relating to external connections:

- a graphical configuration diagram, as illustrated in Fig. 1.
- an alternative representation of the configuration,

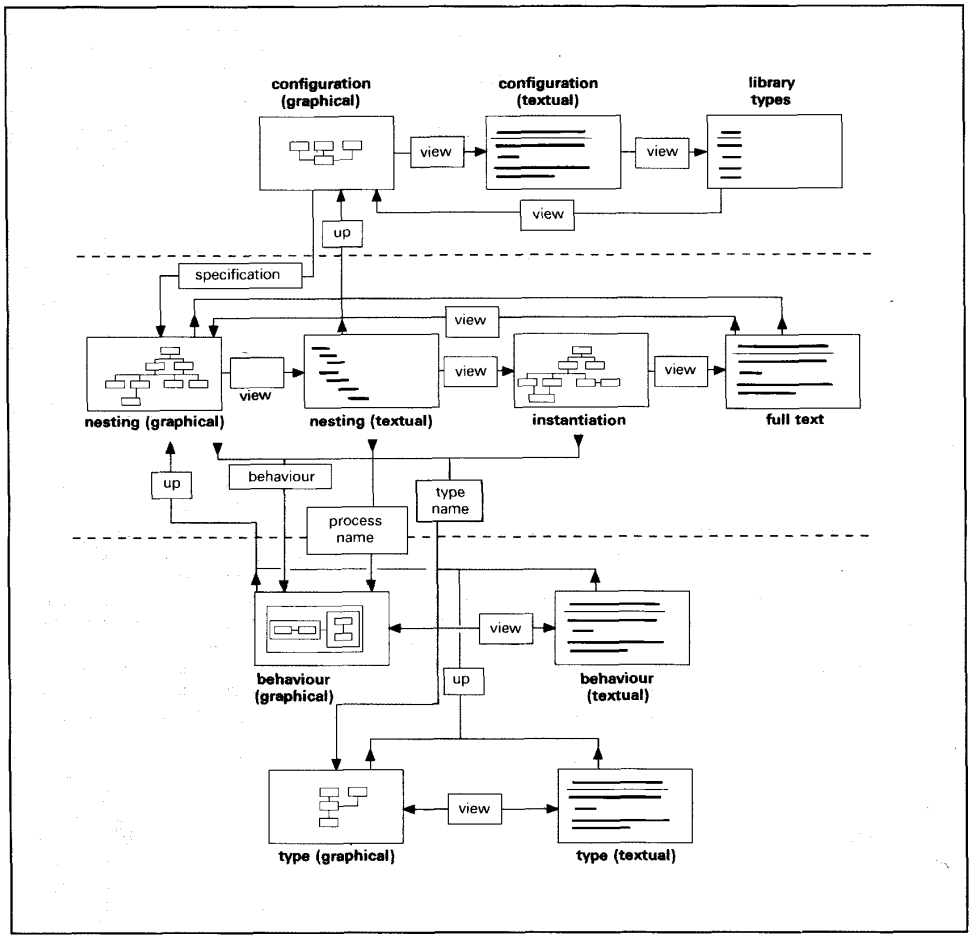

Fig. 2 Specification static view map

showing the information in the specification heading in text form.

- a list of the library types imported by the specification.

The graphical configuration diagram is presented by default, and the other two views can be accessed in the order shown.

The second level in Fig. 2 identifies interpretations of the specification structure. The nesting of process and data type definitions is the default view at this level. It has an alternative textual representation, in which processes and types are named in a list reflecting their definition order; indentation is used to indicate their relative nesting. A third view at this level is a diagram showing the interdependence of process definitions. More specifically, it indicates where one process makes an instantiation reference to another. The final view at this level is the full text of the specification.

The third level shows the structure of individual processes and types. Each has a graphical representation and a textual representation — the full text of the process or type in each case. Type definitions have a diagram identifying other types on which they depend. Process diagrams show the interconnection of events and process instantiations that define their behaviour.

The graphical nesting diagram at level 2 acts as a central view map for level 3. The diagram for the photocopier specification, for example, is shown in Fig. 3. The diagram indicates that the specification defines one process at the top level, *normaloperation*, local to which are two other processes, *copier* and *papertray*. The graphical nesting view may be selected from any other card using the *Nesting* button. Owing to its central role, the nesting diagram is also revealed when the specification box is selected in the top level view, or when the *Up* button is selected from any third-level view.

The graphical representation of individual processes is based on G-LOTOS, the ISO Draft Standard for a graphical version of LOTOS [19]. G-LOTOS diagrams contain all the information present in the equivalent textual representation. In practice, this means that diagrams can contain a substan-

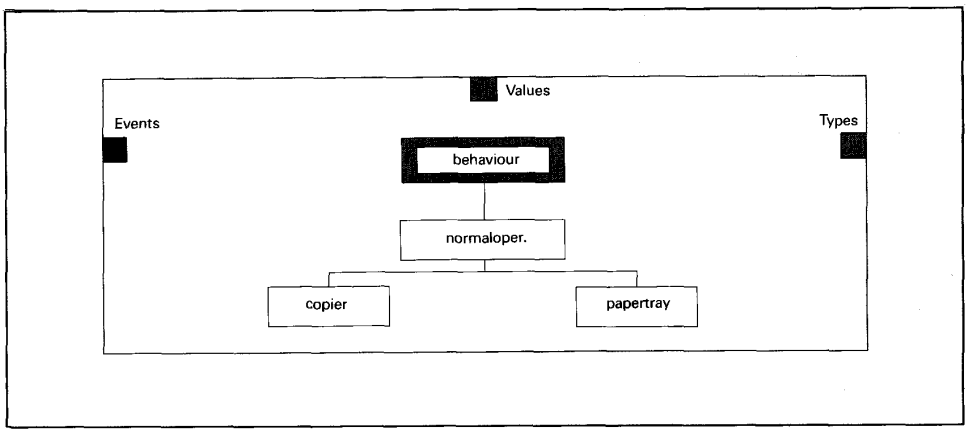

Fig. 3 Photocopier nesting view

tial amount of text. To make the structure of the diagrams more apparent, EXPOSE suppresses certain parts of each diagram, which may then be revealed by selecting the relevant icon. For example, consider Fig. 4, which is a view of the behaviour of the photocopier.

Here, the instantiation of process *normaloperation* is represented by a rectangular box, of standard fixed size, labelled with as much of the process name as will fit. By selecting this component of the diagram, as shown, the full instantiation description is revealed in the elaboration window. Similarly, the photocopier external events may be revealed by selecting the labelled edge connector at the top left-hand side of the view. (Note the *Values* and *Types* boxes are shown as faded to indicate that the specification has neither value parameters nor global types.)

Process behaviour expressions typically make reference to other processes, which means, in practice, that anyone exploring a specification may need to inspect a succession of process behaviour views in order to understand the behaviour of a particular process. Each process view can be

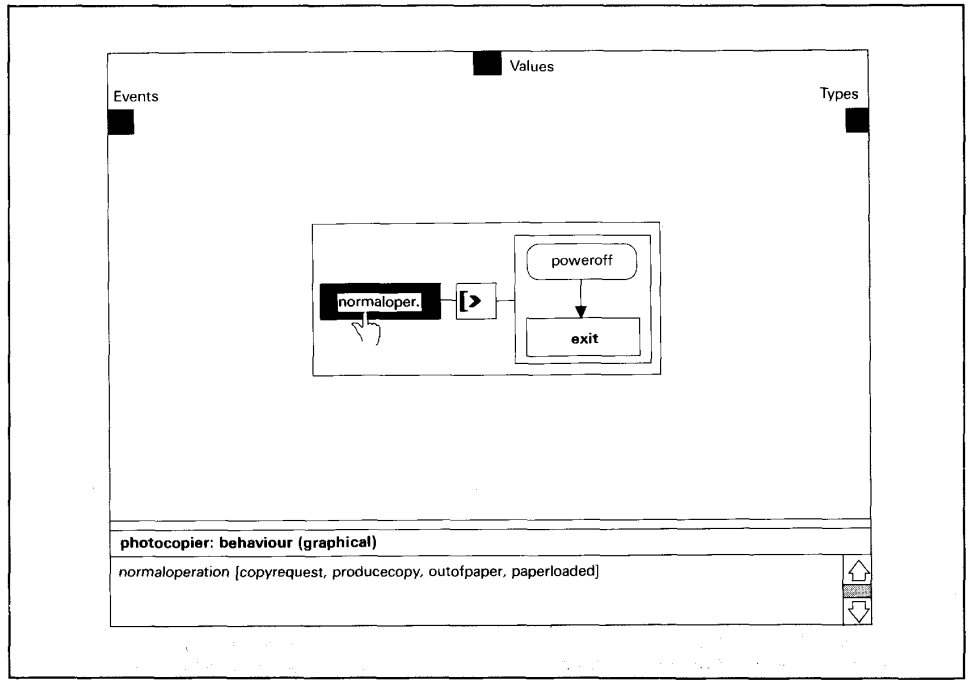

Fig. 4 Photocopier behaviour view

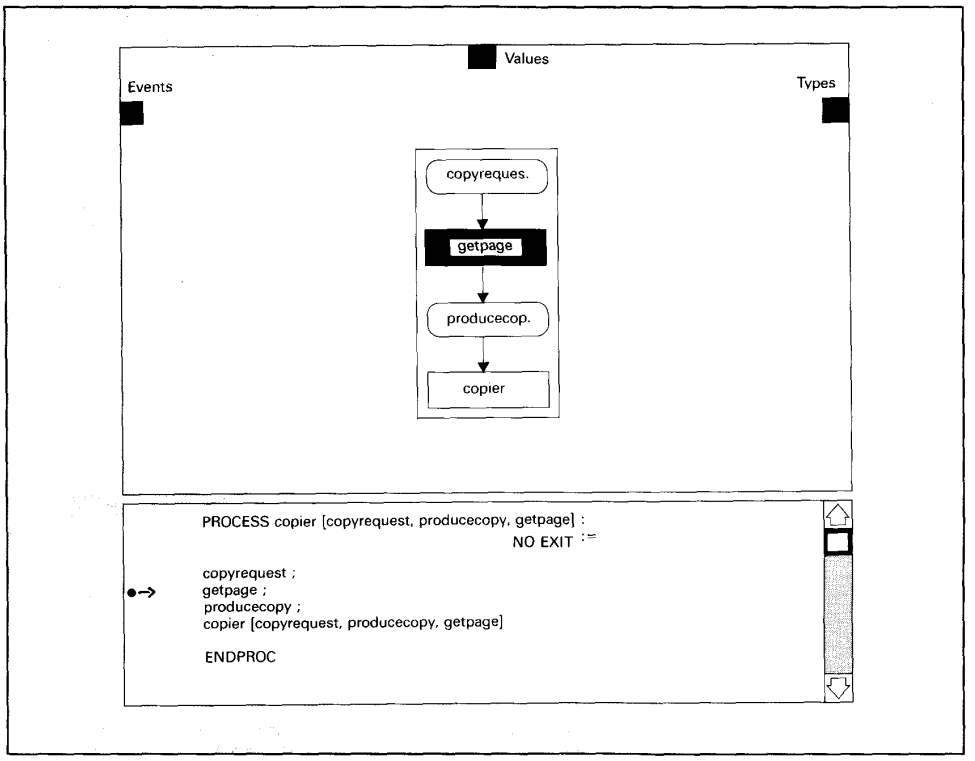

Fig. 5 Dynamic views of copier process

selected from a nesting view, but, as an optimisation, the same selection can be made by clicking on any process identified in a behaviour diagram while holding down the *option* key.

4 Dynamic views

Dynamic views of a specification result from exploring the event sequences defined by that specification. In the initial defined state, there should be at least one, and possibly several, events that can occur. Where there are multiple events, some are independent and others are mutually exclusive. A dynamic view shows *event transitions*. From the initial state, the occurrence of an event leads to a new state defined by the set of events that are permissible. This 'execution' of the specification either proceeds until no further events are possible or the observer decides to stop.

Dynamic views are largely obtained by superimposing representations of *event offers* (the events permitted) and event transitions on static behavioural views. Fig. 5, for example, illustrates the appearance of the graphical and textual representations for the *copier* process at a point where the *getpage* event is offered and able to proceed. In the graphical view, the event icon flashes; selecting this icon (with the *option* key pressed) causes the event transition to occur, and all views to be updated accordingly.

Indications of activity are transmitted up to higher level views. Thus, for example, when the *getpage* event is enabled, both the *copier* process icon (in a behaviour view of the *normaloperation* process) and the *normaloperation* process icon (in the behaviour view of the *photocopier*) will flash.

The state of a process is defined by the events that it offers, and so these must be identifiable by the observer. In circumstances where a process offers an event that cannot proceed, because no other process is offering synchronisation, the corresponding event icon does not flash but instead is shown highlighted.

A dynamic view is obtained by first selecting the *execute* button in the control panel. It is possible to execute the full specification or one of its processes in isolation, and so a menu is shown allowing a selection to be made. Three buttons, *go*, *step* and *abort*, replace the *execute* button. The *go* button causes execution to proceed non-interactively, with the selection of events being determined automatically (randomly) until interrupted by the user. The *abort* button terminates the execution. As an alternative to selecting events directly, the *step* button can be used to bring up a list of permissible events inviting user selection, as illustrated in Fig. 6.

Events in the list are identified by the names that they are given when first defined in the specification. Those events that are passed as parameters to process instantiations may acquire different local names. For example, the *page* event in the *normaloperation* process of the *photocopier* specification has a formal parameter name *getpage* in

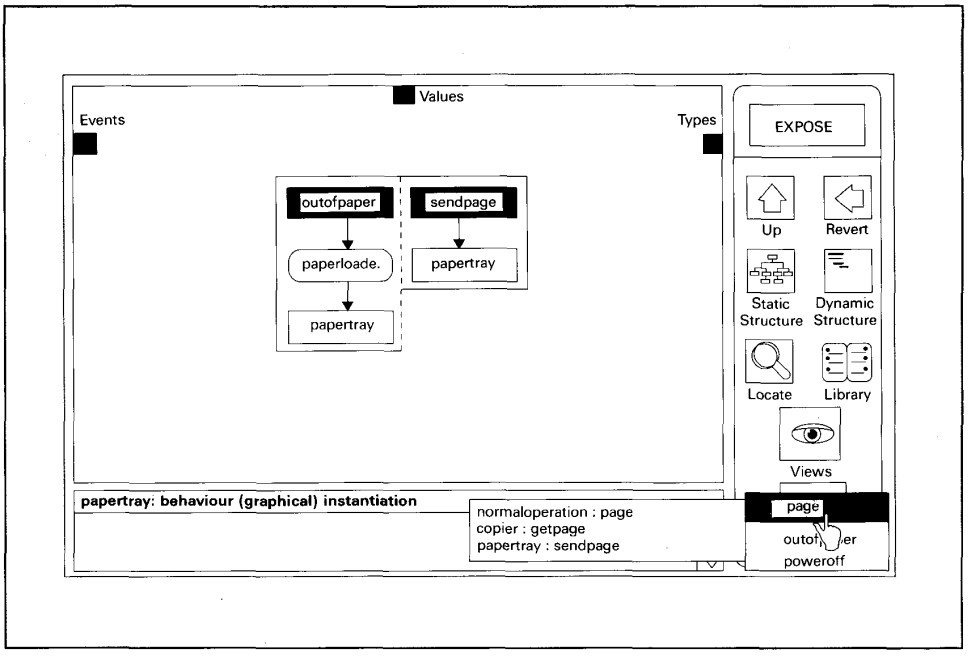

Fig. 6 The event selection menu and descriptive table

the *copier* process and *sendpage* in the *papertray* process. As events may be passed through several instantiations, with effective renaming at each stage, a mechanism is needed to unravel this complexity. The technique used in EXPOSE is to associate a descriptive table with each offered event. This table is presented when an event is selected from the menu with the *option* key depressed. The table contains a list of all the process instances to which the event has been passed, together with the local name used in each case. The descriptive table is also used as a navigational aid; selecting a process instance in the table brings up the graphical view of that instance. In this way, the observer can explore the meaning of any event before selecting it.

A LOTOS specification is a single process, whose execution results in the successive instantiation of other processes interspersed with event transitions. Fig. 7 shows the dynamic views through which this behaviour may be observed. The instantiation view is an indented list, showing the process instantiations that have occurred. Ideally, this view should have a graphical equivalent [2]. However, although such a view was implemented experimentally, it is not the final system because of the unacceptable time required to redraw diagrams dynamically through HyperCard. To avoid an explosion in the size of the instantiation list, tail recursive process references are recognised and suppressed, as suggested in Reference 1. The number of such instantiations is recorded in brackets after each process name in the instantiation list to maintain a full execution history.

Process behaviour is alternatively shown in terms of the trace of events that have occurred; arguably a more general description, since it is independent of the way in which the specification has been constructed.

Dynamic views of particular process instances may be selected from the instantiation list or from the dynamic views in which they appear. As with static views, graphical representations of processes are shown by default and textual views selected via the *view* button.

5 Implementation

EXPOSE is largely written in Pascal Plus [20], a superset of standard Pascal with extensions for modular programming. It is executed as a distributed program on a DEC VAX, running the VMS operating system, and an Apple Macintosh running HyperCard. EXPOSE operates in two phases:

- *view generation*, during which static views of specifications are created.
- *view animation*, during which static views are inspected and dynamic views constructed from the static representations.

The software for both phases has been designed and implemented in a modular fashion, which, apart from the advantage of clarity, security and reusability, allows the isolation of device-dependent facilities, particularly those concerned with graphics. This enables the system to be more easily ported between machines and between different graphics packages. It also facilitiates experimentation with the representation of views.

The basic data-flow diagram for the system is shown in Fig. 8. The input is a sequential text file containing a LOTOS specification, and the output is a HyperCard stack

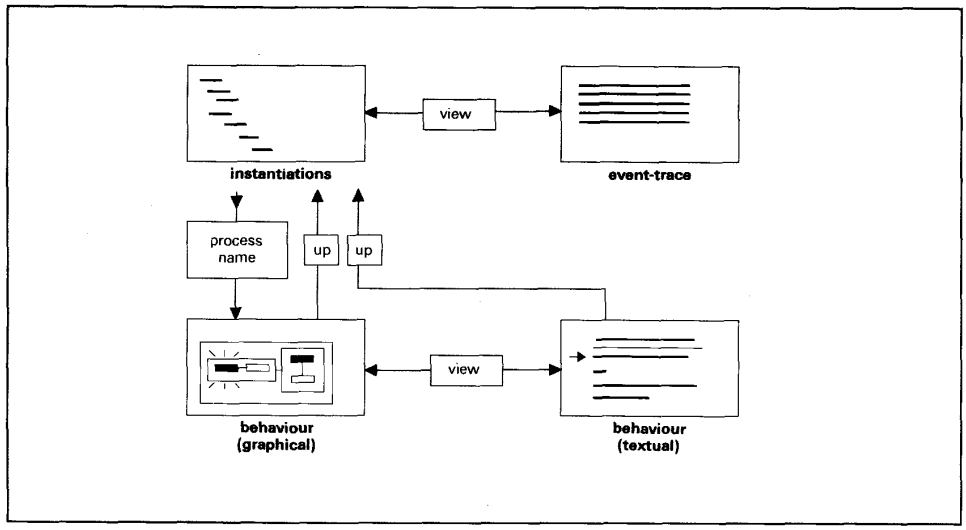

Fig. 7 Dynamic view summary

containing graphical and textual views of the specification. These views are copied and updated by the *animator* to show the behaviour defined by the specification.

5.1 *View generation*

EXPOSE performs a *syntax analysis* on each LOTOS specification that it receives and constructs an equivalent internal representation in tree form. A full listing of the specification is produced with the position and nature of any errors reported. If errors are present, the specification is rejected; otherwise, a *semantic analysis* is performed. The syntax tree is decorated with information describing the nature of all identifiers in the specification, and identifiers that are related are linked. When an error is encountered, a marker is left in the tree, but processing continues so that animation can be performed on those parts of the specification unaffected by the error. Following semantic analysis, a *formatter* puts the specification text into a standard form. The format chosen reflects the graphical representation of the specification and also simplifies the task of animating the textual form.

The formatted text is written to an output file, and the syntax tree updated with information on the location of significant actions identified in the specification, essentially events and process instantiations. From the resulting tree, the *view generator* produces a sequence of HyperCard commands, which, when executed, will create a stack of specification views.

The view generator produces views through various layers of abstraction that turn high-level operations, such as 'draw a process view', into the sequence of drawing commands needed to achieve that effect with the representation chosen. In this way, the representation of the views is isolated, as is the identity of the graphics package in use.

5.2 *View animation*

To initialise animation, an *animator* (on the Macintosh) produces a set of static views from the HyperCard instructions generated in the view generation phase of processing. These views can be examined. When execution of the specification is requested, the *animator* starts up a dialogue with the *interpreter*, which executes the specification and supplies dynamic view information via the *view generator*.

Execution is performed using the LOTOS *transition rules* given in Reference 7. Initially, a copy of the relevant part of the syntax tree is taken to produce the execution tree. The animator uses two basic execution functions. The first, when given an execution tree, returns the set of next possible events (the *initials*), together with information about the processes involved, and the renaming and synchronisation of events within them. This information is used to indicate which events are available in the relevant views and to construct the event-selection menu, as illustrated in Fig. 6. A second function is applied when an event is chosen. The transition rule associated with the selected event is applied, and the execution tree updated by the *interpreter*. In addition, the animation instructions to advance the views to the new state are returned to the *animator*.

6 Evaluation

EXPOSE was developed to explore ways of presenting static and dynamic interpretations of a process-oriented formal specification that might help to make the meaning of such specifications more apparent. It sought to provide clarity, by highlighting the significant relationships among specification components, and was designed specifically to be useful to developers who were already familiar with textual notations. The relationships are given as a set of views, presented in a mixture of graphical and textual representations as appropriate.

A user of EXPOSE submits a textual specification as input, and all views are generated automatically for inspection. In this way, the tool can be judged a success if even

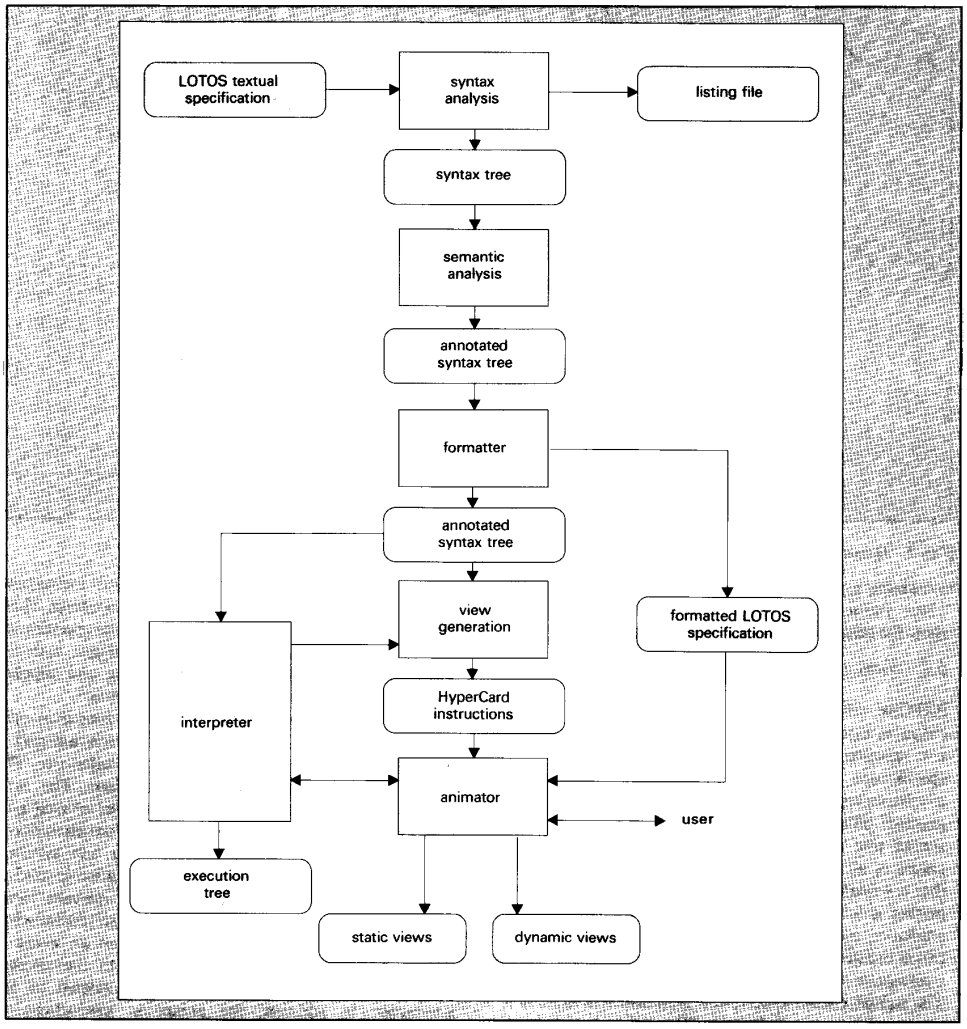

Fig. 8 EXPOSE data-flow diagram

one view is of benefit to the user. Such a conclusion seems very probable, since several of the views provide summary and navigational information that is not present in the textual representation. In many respects, these views have a similar function to the table of contents and the index of a book, two facilities whose usefulness are not in doubt.

Other views give alternative equivalent graphical representations to text, and the value of these views is perhaps debatable. Certainly, there are some people in the formal community who are suspicious of the use of diagrams and think it preferable to 'learn to concentrate attention on the cold dry text of the mathematical formulae, and cultivate an appreciation for their elegant abstraction' [9]. EXPOSE sidesteps this debate by providing equivalent support for text and graphics, although it does show some bias by presenting graphical representations by default.

EXPOSE does not, at present, include full support for data types. They are analysed and static diagrams produced, but a rewriting rule interpreter is required to evaluate data values during the execution of a specification. This limits the application of EXPOSE to so-called Basic LOTOS [7], LOTOS that makes no explicit reference to data, as illustrated in the photocopier example. Work is continuing to complete the implementation.

Based on current experience with EXPOSE, the following additional points can be made.

- Showing dynamic behaviour by superimposing some representation of activity on the static structure of a specification is very successful; the effect is to bring the static description to life. No particular sophistication is required to achieve this result, and the basic HyperCard facilities to

highlight or flash objects on the screen have proved adequate.

- Reformatting the user's text for animation purposes helps maintain a visual link between each textual representation of a component and its graphical equivalent. However, it is recognised that this is not good practice, and it would be preferable for EXPOSE to operate on the original form or to provide the user with an integral editor that allowed the specification to be built in an acceptable form in the first instance.
- Automatic generation of views gives a good first approximation to the layout of various aspects of a specification structure, but, for fine tuning, the user should be provided with some mechanism to edit the resulting diagrams.
- The decision to present views in a clean uncluttered way by hiding details is largely successful. However, displaying the details in a separate elaboration window one item at a time is not always convenient. It would be preferable to have each expansion as a separate object that could be placed anywhere on a view and compressed again when no longer needed.
- Using hypermedia-type links among views provides a convenient browsing facility, but some additional operations are needed to enable experienced users to access some views more directly; this could be achieved by using defined control keys.
- Having access to only one view at a time is very inconvenient when viewing dynamic behaviour. Recent improvements to HyperCard allow several cards to be shown simultaneously, and the use of this facility is being explored.
- The time taken to produce a set of views for a given specification is relatively lengthy. This may be satisfactory when a substantial period is then spent inspecting the views. However, taken as a step in the successive development of a specification, it would be preferable to be able to have access to views immediately after a change. This could be achieved by using a more direct graphical facility than HyperCard, but a better alternative might be to develop and maintain the views in parallel with the text. In this way, small adjustments to the text can be expected to have an equally small effect on views, and so require considerably less processing time than is needed to reconstruct them all.

On balance, the current EXPOSE system has been very beneficial as an experimental tool. In particular, the facilities for static browsing are good, and improvements in HyperCard are likely to make them even better. In the longer term, however, progress must be made towards integrating the construction and animation of specifications to both shorten animation time, and avoid any modifications to the specification representation as developed by the user. Work in this direction is continuing in the SCAFFOLD project, which is funded by the Science and Engineering Research Council and undertaken collaboratively by the University of Ulster, York University and British Aerospace.

7 Acknowledgments

Part of the work described here was supported by the Department of Education for Northern Ireland and by SERC Grant GR/G 03700.

Part of the paper was prepared by David Bustard while on sabbatical leave at the Software Engineering Institute at Carnegie Mellon University in Pittsburgh.

The authors would like to thank colleagues at British Telecom's Research Laboratories in Ipswich, who collaborated on the early animation work from which EXPOSE was derived, particularly Mark Norris and Rodney Orr.

8 References

[1] BUSTARD, D.W., NORRIS, M.T., and ORR, R.A.: 'A pictorial approach to the animation of process-oriented formal specifications', *Softw. Eng., J.*, 1988, **3,** (4), pp. 114–118

[2] BUSTARD, D.W., WINSTANLEY, A.C., NORRIS, M.T., ORR, R.A., and PATEL, S.: 'Graphical views of process-oriented specifications' *in* TURNER, K.J. (Ed.): 'Formal description techniques 88' (North-Holland, 1988)

[3] PATEL, S., ORR, R.A., NORRIS, M.T., and BUSTARD, D.W.: 'Tools to support formal methods'. Proc. 11th Int. Conf. on Software Engineering, Pittsburgh, USA, May 1989

[4] ISO: 'Information Processing Systems — Open Systems Interconnection — LOTOS — a formal description technique based on the temporal ordering of observational behaviour'. ISO 8807, 1989

[5] 'Macintosh Hypercard User's Guide' (Apple Computer, 1987)

[6] GOODMAN, D.: 'The complete hypercard handbook' (Bantam, 1987)

[7] BOLOGNESI, T., and BRINKSMA, E.: 'Introduction to the ISO specification language LOTOS', *Comput. Netw. ISDN Syst.*, 1987, **14,** (1), pp. 25–59

[8] MILNER, R.: 'A calculus of communicating systems', *Lect. Notes Comput. Sci.*, 1980, **9** (Springer-Verlag)

[9] HOARE, C.A.R.: 'Communicating sequential processes' (Prentice-Hall International, 1985)

[10] BERGSTRA, J.A., and KLOP, J.W.: 'Process algebra: specification and verification in bisimulation semantics' *in* HAZEWINKEL, M., LENSTRA, J.K., and MEERTENS, L.G.L.T. (Eds.): 'Mathematics and computer science II' (CWI Monograph 4, pp. 61–94)

[11] VALENZANO, A., SISTO, R., and CIMINIERA, L.: 'An abstract execution model for basic LOTOS', *Softw. Eng. J.*, 1990, **5,** (6), pp. 311–318

[12] EHRIG, H., and MAHR, B.: 'Fundamentals of algebraic specification 1' (Springer-Verlag, 1985)

[13] BRINKSMA, E.: 'A tutorial on LOTOS' *in* DIAZ, M. (Ed.): 'Protocol specification, testing and verification V'. Proc. IFIP Workshop (North-Holland, 1986)

[14] VAN EIJK, P.H.J., VISSERS, C.A., and DIAZ, M.: 'The formal description technique LOTOS' (Elsevier, 1989)

[15] TURNER, K.J. (Ed.): 'Formal description techniques 88' (North-Holland, 1988)

[16] VUONG, S.T. (Ed.): 'Formal description techniques 89' (North-Holland, 1989)

[17] TURNER, K.J.: 'A LOTOS-based development strategy' *in* VUONG, S.T. (Ed.): 'Formal description techniques 89' (North-Holland, 1989)

[18] SHAFER, D.: 'Hypertalk programming' (Hayden, 1988)

[19] ISO: 'Proposed draft addendum to ISO 8807:1988 on G-LOTOS', ISO/IEC JTC1/SC21, 1990

[20] BUSTARD, D.W., ELDER, J.W.G., and WELSH, J.: 'Concurrent program structures' (Prentice-Hall International, 1988)

9 Appendix: The LOTOS language

The LOTOS language consists of two largely independent components:

□ a *process algebra*, based mainly on ideas used in CCS [8] and CSP [9]. This is used to express the temporal behaviour of a system.

□ an *abstract data type* component, based on the algebraic language ACT ONE [12]. 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.

It is possible to construct some specifications using only the process algebra component of the language. This restricted form is usually referred to as *basic LOTOS*. The discussion below concentrates on this form.

In general, a system is described in LOTOS in terms of a hierarchy of interacting *processes*. Syntactically, this is expressed using a structure of nested process definitions, each one having a scope in much the same way as procedures in declarative programming languages.

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, and this 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 can be passed to processes through parameters. A process can be recursively instantiated to specify repeated behaviour.

There are two basic processes built into LOTOS: **stop** and **exit**. These represent inactivity and successful termination, respectively. A special event δ is implicitly offered by **exit**.

A special event $\boldsymbol{i}$ 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

- 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 → set of events

- 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 also given, followed by the definition of the *initials* function derived from them. These have been used as the basis for the implementation of the basic LOTOS interpreter used by the EXPOSE animation system. In the discussion, the following symbols are used:

$\in$, $\notin$, $\cup$, $\cap$ = the set operators for inclusion, exclusion, union and intersection
$\varnothing$ = the empty set
B, $B1$, $B2$ = behaviour expressions
$g \in G$, where G is the set of user-defined actions
$\boldsymbol{i}$ = the unobservable internal action
$\mu \in Act$, where $Act = G \cup \{\boldsymbol{i}\}$, i.e. the set of explicit actions
$S = [g_1 \cdots g_n]$, a finite set of user-defined action names
δ = successful termination
$g^+ \in G^+$, where $G^+ = G \cup \{\delta\}$, i.e. the set of observable actions
$\mu^+ \in Act^+$, where $Act^+ = Act \cup \{\delta\}$, i.e. the set of all actions
g/g' = the replacement of occurrences of the name g' by g
$\phi = [g_1/g'_1 \cdots g_n/g'_n]$ = 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**.

exit $\xrightarrow{\delta}$ **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

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.

$\mu; B \xrightarrow{\mu} B$

initials $(\mu; B) = \{\mu\}$

choice ([ ])

$B1$ [ ] $B2$ means that either the behaviour $B1$ or $B2$ can occur. The outcome depends on the events offered by the environment, unless the initial events of $B1$ and/or $B2$ are identical or involve the internal event $\boldsymbol{i}$. In this case, the choice is non-deterministic between $B1$ and $B2$. 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 sub-expression occurs to produce a resulting behaviour expression (as shown above the horizontal line), the overall construct will perform the same action to produce the same resulting expression (shown below the line):

$$\frac{B1 \xrightarrow{\mu^+} B1'}{B1 \;[]\; B2 \xrightarrow{\mu^+} B1'}$$

$$\frac{B2 \xrightarrow{\mu^+} B2'}{B1 \;[]\; B2 \xrightarrow{\mu^+} B2'}$$

The initials of a choice expression is the union of the initials of the individual sub-expressions:

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

parallel composition (|[a, b, ...]|)

$B1$ |[a, b, ...]| $B2$ means that $B1$ or $B2$ occur in parallel and share, or synchronise 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 $B1 \parallel B2$. The implicit event in successful termination δ is always shared between parallel processes. This means that behaviours 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 expresses those that are ($g^+ \in S \cup \{\delta\}$):

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

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

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

$$\text{initials}\,(B1 \mid S \mid B2) = (\text{initials}\,(B1) - S)$$

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

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, 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 (≫)

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

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

$$\frac{B1 \xrightarrow{\delta} B1'}{B1 \gg B2 \xrightarrow{\boldsymbol{i}} B2}$$

$$\text{initials}\,(B1 \gg B2) = \text{initials}\,(B1)$$

hiding (**hide** ··· **in**)

Hiding makes named events internal to a behaviour expression, and so unavailable for interaction with the environment, essentially giving them the characteristics of the internal event $\boldsymbol{i}$. Inference rules are given as follows:

$$\frac{B \xrightarrow{g} B',\ g \in \{g_1 \cdots g_n\}}{\textbf{hide}\ g_1 \cdots g_n\ \textbf{in}\ B \xrightarrow{\boldsymbol{i}} B'}$$

$$\frac{B \xrightarrow{\mu^+} B',\ \mu^+ \notin \{g_1 \cdots g_n\}}{\textbf{hide}\ g_1 \cdots g_n\ \textbf{in}\ B \xrightarrow{\mu^+} B'}$$

$$\text{initials}\,(\textbf{hide}\ g_1 \cdots g_n\ \textbf{in}\ B) = \text{initials}\,(B)[i/g_1 \cdots i/g_n]$$

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 more manageable units. Its use also allows the parameterisation 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 $P[g'_1 \cdots g'_n] := B_p$ ***endproc***

process instantiation is expressed by the inference rule

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

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

$$\frac{B \xrightarrow{g'} B',\ \phi = [g_1/g'_1 \cdots g_n/g'_n],\ g/g' \in \phi}{B\phi \xrightarrow{g} B'\phi}$$

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

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:

$$\text{initials}\,(P[g_1 \cdots g_n]) = \text{initials}\,(B_p)[g_1/g'_1 \cdots g_n/g'_n]$$

Adam C. Winstanley is with the Department of Computer Science, Queen's University of Belfast, Belfast BT7 1NN; and David W. Bustard is with the Department of Computing Science, University of Ulster, Coleraine BT52 1SA.

The paper was first received on 20th September 1990 and in revised form on 9th April 1991.