INTERACTORS: FORMAL ARCHITECTURAL MODELS OF USER INTERFACE SOFTWARE

1 Introduction

This paper investigates the application of formal methods for the design, specification and verification of interactive systems. It argues that the practical application of formal techniques requires that interactive systems be modelled at an architectural level of abstraction which is relevant to the concerns of the software developer. The resulting models are generically termed *interactor* models. This paper introduces interactor models, discusses such a model extensively and examines the limitations of the approach and its links to current trends in the field of user interface software design and development.

2 Using formal specification in the design and development of user interfaces

Traditionally, the envisaged role of formal specifications is to describe the essence of a system function without premature commitment to implementation details. The underlying philosophy of applying formal methods in the field of human-computer interaction is that by applying generic user interface design principles and rigorous reasoning, usable and effective systems may be designed by an almost mechanical process of refinement or transformation of specifications (Harrison and Dix 1990). Formal specifications are then seen to have an integrative role: they bring together contributions originating from different perspectives and allow the developer to check their consistency and to identify issues which require further consideration (Duke and Harrison 1994).

The following limitations of this view are suggested:

- A formal specification used as a 'repository' of all interface design decisions makes the design process revolve around authoring, checking and updating the specification. This approach contrasts the need for rapid prototyping and end-user involvement in the design process: articulating any interface (re)design decision is more costly and the product of this activity is obscure to the end-users of the system (contrast a formal specification to a prototype of the designed user interface). Further, formal specifications typically abstract away from issues of great impact on the usability of the product, e.g., layout design, aesthetics, and contextual issues typically captured in rich representations such as scenaria or prototypes. This tension, between formal methods and mainstream approaches for the design and development of user interfaces, suggests that rather than dictating methods and processes for software development, formal methods researchers should attempt to find out how their methods and tools may help within currently established approaches. The design of an interactive system should not be driven by a formal method but may use it as a means of representing designs and assessing decisions.
- Because of the inherently abstract nature of formal specifications, a single specification cannot capture all the issues which impact the quality of the design system, indeed many interesting insights are obtained by combining several modelling techniques, some of them informal by nature. A formal specification is good for a specific job and for a well defined scope of the problem domain. Not all design decisions but, in fact, just a few may be based on the formal specification.
- For any application domain, the use of formal methods relies on identifying and formalising appropriate abstractions. This task is far from trivial. The applicability and the utility of formal methods can be enhanced if re-usable generic abstractions for modelling interactive software and a library of 'off the shelf' generic properties are identified. Checking a system specification against these generic properties can act as a 'safety net' during its design.

This perception of the role of formality suggests the type of research discussed hereby. The first aim is to identify re-usable and generic abstractions of user interface software and formalise generic properties for their verification and validation. Formal refinement is not an important concern in user interface software development. Significant gains can be expected by the use of a standard specification language, off-the-shelf tool support, and by the modularity and re-use of specifications.

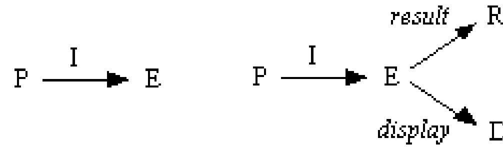

Figure 1. The PIE and red-PIE models (Dix 1991). Arrows represent mappings between sets.

Finally, an important consideration is to relate interface specifications to other modelling approaches in human computer interaction.

3 Abstract models of interactive systems

Early and influential approaches to modelling interactive systems and their properties can be described as *abstract models of interactive systems* (abstract models for short). These were developed in an effort to formalise heuristics/principles for the design of interactive systems. A well known and documented abstract model is the PIE model (Dix, 1991) discussed below in brief.

The PIE model describes an interactive system as a 'black box' to which input is given and of which output may be observed. The model relates user *programs* P, which are sequences of commands, to the *effects* E they have, via an *interpretation* function I:P→E, (see figure 1). The effects may refer to the display, the entire information managed by the interactive system, etc. Properties of interaction are expressed abstractly in terms of relations between these sets, without referring to any internal representation of the system. The PIE model affords concise definitions of concepts such as:

- *Predictability*: can the user predict the effect of his commands by the externalised behaviour of the system?
- *Obsevability*: how much of the internal state of the system can be observed by the user at any moment, or through interacting with the system?
- *Reachability*: From a given state (or given states) of the system is it possible to reach some target state (or target states)?
- *Determinism*. Is the effect of a command deterministically defined by the context of its application?

These concepts are classes of properties which are made more concrete by explicating what scope of the system they apply to, what sets of commands or states they refer to. As an example of how PIE formalises such properties we examine predictability.

A system can be characterised as predictable if for any two programs p and q that have the same effect, also their extension by the same program r will have the same effect (Dix 1991):

$$\forall p, q, r : P | I(p) = I(q) \bullet I(pr) = I(qr)$$

PIE has been extended to the red-PIE model to express the relationship between the *display* (what the user sees) and the *result* (what the user wants to achieve through interacting with the system). The red-PIE model includes the definition of *result*, a mapping from the effect space E to the result-space R and *display*, a mapping from E to a display-space D (see figure 1).

A system is called observable if two programs that produce identical displays also produce identical results: :

$$\forall p, q : P | display(I(p)) = display(I(q)) \bullet result(I(p)) = result(I(q))$$

Note that observability describes the theoretical feasibility of observing the state of a system, rather than the user's ability to perceive and process this information. This observation applies also to predictability and to similar expressions of usability related system-properties discussed throughout this paper.

The PIE family of models has exercised significant influence on later formal models of interactive systems, but it is very hard to apply in practice. In particular, the following shortcomings can be associated with this model:

- PIE gives an unstructured description of the interface which does not portray adequately the structure of the interaction, or a constructive definition of the user interface software.
- The directedness of the interpretation mapping I does not portray how I is sensitive to the display contents D, and does not help model the re-use of output as input, which is an essential aspect of graphical user interfaces.
- PIE does not help write specifications of interactive systems as compositions of smaller sized PIE descriptions. For engineering purposes a constructive model is required, that will allow the description of the user interface as a composition of formally specified autonomous units.
- While a range of interaction properties can be expressed in terms of the PIE family of models, the abstractions discussed are sometimes hard to grasp, as they do not relate directly to the software entities that will be developed. It requires considerable expertise to select which properties should be set as requirements upon a particular system and how to interpret their abstract and general expressions for a particular design problem.

Considering the limitations of abstract models identified in this section, we can refine our view of applying formal methods to user interface design and development. The work presented in this article, focuses on abstractions of user interface software architectures. These architectures are well documented, tested and widely used. The resulting formal models are generic and re-usable in the particular domain and they are *architectural* in nature, i.e., the individual abstract entities can map one-to-one to software components and their formal composition corresponds to the architectural composition of elementary software units. Such formal models are instantiated to a formal specification of a user interface that can be used for analytical purposes or, less likely, as a blueprint for the implementation of the user interface. Authoring formal specifications is supported by the definition of formal specification templates for modelling user interface software and specifications of 'off the shelf' generically applicable properties such as predictability and observability discussed earlier. For practical reasons, a standard specification language and general purpose tool support are used rather than purpose made notations and tools.

4 Architectures for user interface software

An *architectural model* (or *pattern*):

> '…expresses a fundamental structural organisation schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities and includes rules and guidelines for organizing the relationships between them.' (Buschmann, Meunier, Rohnert, Sommerlad and Stal 1996).

Such a model embodies software design expertise pertaining to the structure of a particular class of systems and, ideally, 'packages' it and encourages its re-use. The model may be instantiated to derive concrete system architectures. This instantiation may be supported by software tools, in which case the model is an *implementation architecture model.* Alternatively, it may be supported by a set of heuristics or conceptual aids for designing an architecture or directly a software system, in which case the model is a *conceptual architecture model.* This section discusses two software architecture models specific to graphical user interfaces.

Most software architectures for graphical user interfaces distinguish software components with respect to the functionality they support:

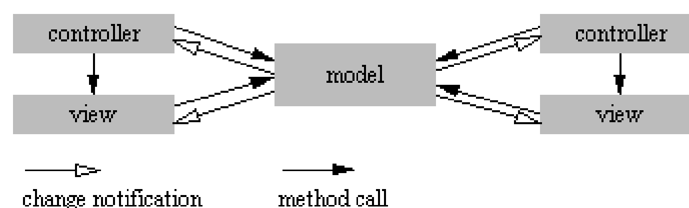

Figure 2. A model and two view-controller pairs.

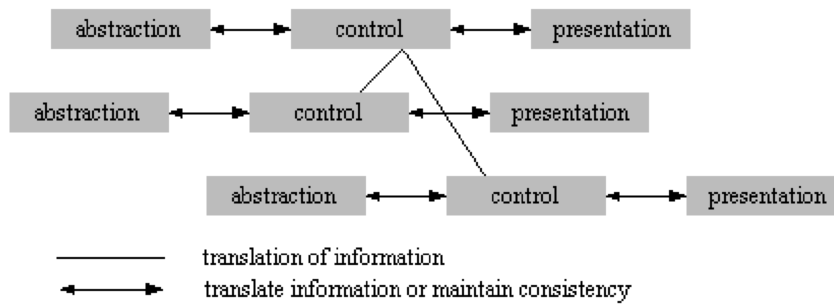

Figure 3. The PAC model (Coutaz 1987) structures the interface system as a composition of PAC agents, but does not prescribe a particular communication and control mechanism.

- The application-specific functionality,
- the dialogue structure of interaction, i.e. the temporal sequencing of individual interaction steps and
- the appearance / display of the interface.

This partitioning aims to ensure the separation of concerns for the software designer/developer and the independence between components. These components may be realized as software layers or they may be constituent components for objects in an object-based architecture. Object-based architectures model the interface software as a composition of co-operating objects. This has several advantages concerning iterative design, support for distributed applications and support for multi-threaded dialogues.

Model-View-Controller (MVC) by Krasner and Pope (1988) is the most widely known implementation architecture model. The user interface is formed as a collection of objects of three types:

- The model of MVC is responsible for managing application-specific data and functionality.
- The view is responsible for displaying the model data (or part of it).
- The controller, is associated with a view and is responsible for handling user input.

MVC emphasises the independence of the model from views and controller. A single model may have many view-controller pairs as 'dependants', as shown in figure 2. The controller, is specific to a single view. It handles user input and modifies its model via a call-back method. The model may receive input from any object in the system via a method call. When its value changes it notifies all its dependants that it has changed and it is up to them to update themselves accordingly.

The Presentation-Abstraction-Controller (PAC) model by Coutaz (1987), is an influential conceptual architecture model which structures an interactive system as a hierarchical composition of PAC triads; the hierarchical organisation of these triads is illustrated in figure 3. Each triad has the following components:

- An abstraction which holds the application specific functionality and is similar to the model of MVC,
- a presentation component which handles input and output, and which corresponds to a view-controller pair of MVC,
- a controller component which maintains the consistency of the two and implements the hierarchical composition of PAC triads and the communication between them.

PAC emphasizes the hierarchical structure of interactive software and provides guidelines for structuring the composition hierarchy. However, it is very vague regarding the relationships between the presentation, abstraction and control sub-components, and regarding their communication and behaviour.

Coutaz (1997) identifies the need to formalise user interface architectures to help communicate clearly concepts that normally rely on intuitions and knowledge shared amongst software engineers. This work adopts a symmetric, but consistent, point of view: formal abstractions of interactive software must model the software architecture to render the formal specification and verification of interactive software a valid and useful tool for software designers. The formal model discussed in the remainder

of the paper does not aim to formalise any particular software architecture model. It reflects common traits of user interface architectures so that the formal model can be used as a basis for the specification of user interfaces that are built following PAC, MVC, or other similar software architectures.

5 The concept of an interactor

The term *interactor* refers to software objects that have a display and, in general, support both input and output. MVC or PAC triplets are such interactors. An interactor can describe a software entity at any abstraction level, e.g., a cursor, a dialogue box or even a whole application. In practice, it is useful to consider the scope of the interactor model as shown on figure 4. At the lower level we assume the existence of input event management, e.g., to direct user inputs to the appropriate interactor, and display management, e.g, for ensuring repainting damaged areas on the screen, for providing some graphical output capabilities, etc. These capabilities are typically provided by window systems.

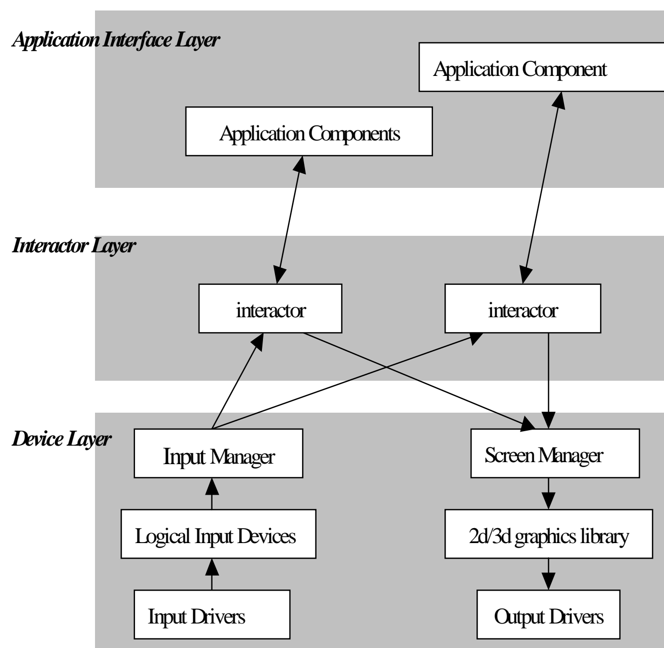

Figure 4. Scope of interactor models.

Formal interactor models are abstractions, which are used to model interactive systems as compositions of independent entities. In the remainder of the paper, the term interactor refers to a formal interactor model. Faconti (1993) defines an interactor as:

> '...an entity of an interactive system capable of reacting to external stimuli; it is capable of both input and output by translating data from a higher level of abstraction to a lower level of abstraction and vice versa.'

This definition considers the user interface as a layered composition of interactors, mediating between a user and the functional core of an interactive system. Each layer of interactors (or a single interactor) distinguishes two levels of abstraction for the data flowing between the user and the functional core. The input functionality of the interactor is to raise the abstraction level of the data it receives and the reverse holds for output which is communicated to the user.

6 The Abstraction-Display-Control (ADC) Interactor Model : Informal Description

The ADC model distinguishes three aspects of an interactor in analogy to the functional partitioning of MVC and PAC. These are:

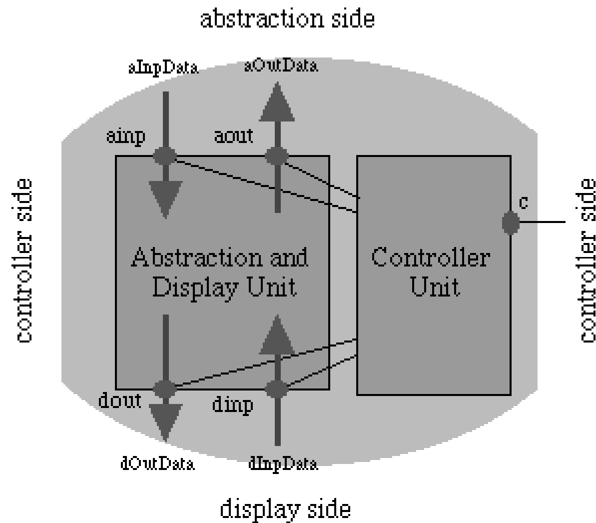

Figure 5. The internal structure of an ADC interactor.

- The Abstraction models data stored and managed by the interactor, e.g., in order to provide input to the application or to other interactors.
- The Display models data which is output either directly to the screen or indirectly through other interactors.
- The Control defines the dialogue, i.e., the sequencing of interactions with the environment that this interactor supports.

The Abstraction and the Display data are encapsulated within a dedicated component, called *the Abstraction-Display Unit* (ADU). The ADU provides a set of operations for reading and modifying the Abstraction and the Display. It is akin to the classical software engineering notion of an 'object'.

Operations on the data are effected through interactions on the *gates* of the ADU. Gates are architectural elements which 'localise' and group interactions with the environment of the interactor. The ADU is 'neutral' with respect to dialogue, i.e. it is always ready to receive input and to send output on all its gates. The actual dialogue of the ADC is modelled separately in the Controller Unit (CU), i.e., interactions on the gates of the ADU are enabled/disabled by the CU.

Figure 5 illustrates the ADC interactor as a barrel-shaped node featuring arches at the top and bottom and two straight sides. The top arch is referred to as the a*bstraction side*, the bottom arch the *display side* and the vertical sides are called the *controller sides*. Gates are shown as arrows when corresponding interactions communicate data or as lines when no data communication takes place. Interactors drawn as barrels will be linked by lines on their gates, to represent the communication between them. Figure 5 also features the internal structure of an ADC interactor as a composition of the ADU and the CU. This structure is common to all ADC interactors so it is omitted in most diagrams.

The direction (inwards or outwards pointing arrow) and the side of gates in the diagram illustrate its *role* for the interactor. The role of the gate refers to the type of interactions it localises:

- *dinp* for graphical input, i.e. input whose meaning is sensitive to the display content,
- *dout* for graphical output,
- *ainp* for non graphical input, e.g., some value sent by the application or some other interactor,
- *aout* for non-graphical output, e.g., some value sent from the interactor to the application or to another interactor, and *c* for control interactions,
- *c* for control (syncrhonisation only) interactions.

For example, an interaction on a *dout* gate will update the display but should not affect the abstraction. A graphical input, e.g., from a pointing device, will have an effect on both the display state - because of some instantaneous feedback - and to the abstraction state, e.g., by interpreting the effect of pointing.

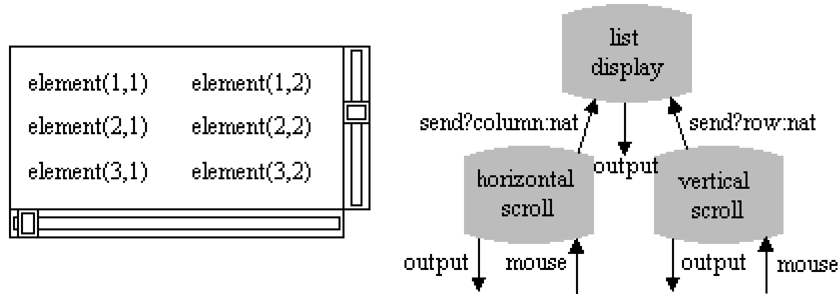

Figure 6. A scrollable list as a composition of three interactors.

A user interface can be modelled as a composition of interactors. This composition is illustrated by connecting interactors on their gates to illustrate data communication or synchronisation between interactors. For example, figure 6 shows a static list of strings which is displayed as a two-dimensional array on a window. Two sliders allow the displayed portion of the list to be scrolled in two dimensions. Each slider receives input from its display side (assuming some pointing device), it provides feedback directly on its display and it interprets its input to instruct the list interactor to scroll, e.g., by passing some integer value indicating the amount and the direction of scrolling.

Interactors can model small-scale interactive components e.g., the sliders above, buttons, menus, etc. but the model applies also to higher level entities in which case the abstraction level of input and output operations is correspondingly raised. For example, an interactor can represent a text editor. An interaction by the user may be, say, to select the name of a file to edit and the output may be an on-screen representation of the file contents. The file selection may be supported by an interactor like the scrollable list of figure 6, but the text-editor can abstract away from this level of detail.

The ADC model emphasises the architectural elements of the interactor: its gates, their role, their grouping to sides and the composition of interactors to form complex interface specifications. It treats separately dialogue and state specifications. Compared to the informal software architecture models discussed of section 4, the ADC model is more abstract as it does not prescribe a particular configuration for composing interactors, e.g., a tree structure. Further, it abstracts away from the mechanism by which display and abstraction data are kept consistent within a single interactor, e.g., the notify-update mechanism of MVC.

7 Formal specification of ADC interactors

This section discusses the formal specification of ADC interactors using the LOTOS language (ISO 1989). The semantics of LOTOS are summarised in the appendix. In general, an ADC interactor is specified as a LOTOS process formed by the parallel composition of processes ADU and CU:

```
process ADC[Gc∪Gio]:noexit :=
  ADU[Gio](a,ds,ds) |[Gio]| CU[Gc∪Gio]
endproc
```

We note the following:

- LOTOS processes specify interactions on a set of gates, which are normally listed between square brackets. For conciseness and generality we indicate sets of gates in the behaviour expression above. G_{io} stands for all the gates used for input and output, and G_c stands for the remaining gates used by the controller (see figure 5).
- Parallel composition is denoted by `|[…]|`. Between the square brackets we list the gates over which the two processes synchronise. In the behaviour expression above, the processes synchronise at all the gates of the ADU.
- The ADU has three state parameters listed between round brackets. Their purpose is discussed later in this section.
- The ADU interacts with its environment through a set of input-output gates G_{io}. Interactions on these gates invoke operations to read or modify the state parameters. These operations are specified in an interactor specific data type, the Abstraction-Display data type (AD).

- CU specifies the temporal ordering of interactions on all the gates of the interactor: the input-output gates G_{io} and the control gates G_c.

SPECIFICATION OF THE DATA TYPE AD

The signature for the generic data type AD is as follows:

```
type ad is
sorts
  abs, disp, dInpData, aInpData, aOutData
opns
  input: dInpData, disp, abs -> abs
  echo: InpData, disp, abs -> disp
  render: disp, aInpData -> disp
  receive: abs, aInpData -> abs
  result: abs -> aOutData
endtype
```

- The sorts *abs* and *disp* store the state parameters, *dInpData* describes input arriving at the display side (gate *dinp*), *aInpData* describes input arriving at the abstraction side (gate *ainp*), and *aOutData* describes data which is sent to the 'clients' of the interactor (through gate *aout*).
- Operations *input* and *echo* describe the interpretation of graphical input to update the abstraction and the display sorts of the interactor. Graphical input is given a special treatment because its interpretation depends on the current content of the display. This dependence is modelled by having the current display state as an argument of operations *input* and *echo*.
- Operations *receive* and *render* describe the interpretation of non-graphical input, to produce a new abstraction and a new display state.
- *Result* is an inquiry operation that calculates a value which will be input to other interactors or to an application.

For a given interactor, the abstract data type AD will have interactor-specific sorts and operations. The data type AD can have a variable number of sorts and for each sort a variable number of operations of the above categories, depending on the gates and the sort of data communicated over a gate. Thus, AD defines for all ADC interactors a classification of the data it manages and of the operations upon the data. It is not necessary to name operations as 'input', 'echo', etc., in all data types associated with the interactors. Rather, there is a commitment that all operations have the purpose and syntax of the operations defined here. This defines a consistent specification style for the data types associated with all ADC interactors. The signature described here can be extended with equations to describe the relationship between the abstraction and the display sorts and the semantics of the operations specified.

SPECIFICATION OF THE ADU: LINKING BEHAVIOURAL AND DATA SPECIFICATIONS

The operations of data type AD are linked to interactions by the ADU. The ADU offers a choice (denoted by the choice operator []) between interactions on any of its gates. With each such interaction, the ADU instantiates itself recursively and modifies its state parameters by applying the corresponding operations upon them.

```
process ADU[dinp, dout, ainp, aout](a:abs, dc,ds:disp):noexit:=
  dinp?x:dInpData; ADU[dinp, dout, ainp, aout](input(x,ds,a),echo(x,ds,a),ds) []
  dout!dc; ADU[dinp, dout, ainp, aout](a,dc,dc) []
  ainp?x:aInpData; ADU[dinp, dout, ainp, aout](receive(a,x),render(dc,x),ds) []
  aout!result(a); ADU[dinp, dout, ainp, aout](a,dc,ds)
endproc
```

Process ADU has two state parameters holding values of sort *disp.* Parameter *ds* holds the last value output on the display from gate *dout.* Parameter *dc* is the computed display which the interactor will output on the next interaction on gate *dout.* At any instant in its lifetime the ADU does not necessarily display its most recently calculated display-data, so parameter *ds* is necessary to maintain a separate record of the value of the display that is used to interpret all arriving user input.

The ADU maps interactions on its gates to corresponding operations of the data type AD. For example, if there are several graphical input gates, e.g. press, click, point, then the ADU will apply for

each a corresponding input and an echo operation to update the abstraction and the display. The CU can be any LOTOS process which synchronises with the ADU and which does not have any state parameters to read and store data.

EXAMPLE

A slider is specified here as an ADC interactor. The slider is a small but interesting example, because it is indicative of how to model direct manipulation interfaces and there is a clear distinction between its abstraction and display data. A larger scale example can be found in Markopoulos (1997) which concerns the specification of the graphical interface to a multimedia application. Figure 7 illustrates a typical slider interactor. The slider can provide random access to an indexed set of data, e.g., a sequence of characters, or static images composing a movie. It is easier to specify *sldr_ad* in a modular fashion. The index is specified as an abstract data type *boundedVal* and its graphical presentation is specified by data type *sliderBar*. For the sake of introducing *sldr_ad* we discuss below a very rudimentary specification of these data types.

BoundedVal supports three inquiry operations that return the upper and lower bounds and the value that ranges between them, and operations to set this value.

```
type boundedVal is Integer
sorts
 boundValue
opns
 lo, val, hi: boundValue -> Int
 setVal: boundValue, Int -> boundValue
 incr, decr: boundValue -> boundValue
eqns
 forall b:boundValue, n: Int
ofsort Int
 val(setVal(b, n)) = n;
 lo(setVal(b,n)) = lo(b);
 hi(setVal(b,n))= hi(b);
ofsort bool
 lo(b) le val(b) = true;
 val(b) le hi(b) = true;
ofsort boundValue
 incr(setVal(b,n)) = setVal(b,s(n));
 decr(setVal(b,n)) = setVal(b,p(n));
endtype
```

A simple and abstract model of a slider bar graphic is characterised by two entities: a rectangle and a point. Let *sliderBar* extend a data type *Graphics* (omitted here) which models such basic graphical entities. The operations of *sliderBar* may change or return the value of the rectangle and the point.

```
type sliderBar is Graphics
sorts
 sliderBar
opns
 mksliderBar: rct, pnt -> sliderBar
 changePnt : sliderBar, pnt -> sliderBar
 changeRect: sliderBar, rct -> sliderBar
 rect: sliderBar -> rct
 point: sliderBar -> pnt
eqns
 forall p:pnt, r:rct, sb:sliderBar
ofsort rct
 rect(mksliderBar(r,p)) = r;
 rect(changeRect(sb,r)) = r;
 rect(changePnt(sb,p))=rect(sb);
ofsort pnt
 point(mksliderBar(r,p)) = p;
 point(changeRect(sb, r)) = point(sb);
```

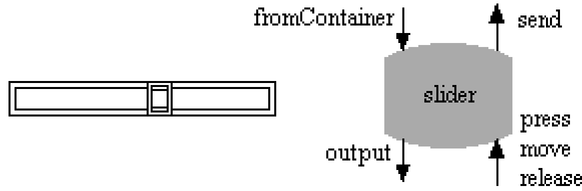

Figure 7. The slider interactor and its illustration as an ADC interactor.

```
 point(changePnt(sb,p))=p;
endtype
```

Sldr_ad can now be defined by extending *sliderBar* and *boundedVal* and by adapting the generic signature for data type ad shown earlier:

```
type sldr_ad is sliderBar, boundedVal
opns
 input: pnt, sliderBar, boundValue -> boundValue
 echo: pnt, sliderBar, boundValue -> sliderBar
 render: sliderBar, rct -> sliderBar
 receive: boundValue, rct -> boundValue
 result: boundValue -> Int
eqns
forall r:rct, p:pnt, sb: sliderBar, bv: boundValue
ofsort boundValue
 receive(bv,r) = bv;
ofsort sliderBar
 echo(p,sb,bv) = changePnt(sb,p);
 render(sb,r) = changeRect(sb,r);
ofsort Int
 result(bv)=val(bv);
endtype
```

Operations *receive* and *render* specify the effect of resizing the enclosing window. Operation *result* specifies the extraction of an integer index from the *boundValue* abstraction of this interactor.

The process modelling the interactor slider is specified as follows (let BV and SB be initial values for the state parameters):

```
process sldr[press, move, release, output, fromContainer, send] :noexit :=
  sldrADU[press, move, release, output, fromContainer, send](BV, SB, SB)
         |[press, move, release, output, fromContainer, send]|
  sldrCU[press, move, release, output, fromContainer, send]
endproc
```

The ADU of the example has three input gates on the display side where *press*, *move* and *release* mouse events are input. These interactions are associated with the input of a point, which is used as an argument by operations *input* and *echo*. The value returned by operation *result* is output on gate *send*. The computed value of the display is output on gate *output*.

```
process sldrADU[press, move, release, output, fromContainer, send]
     (a: boundValue, dc, ds: sliderBar) : noexit :=
press?x:pnt;  sldrADU[...](input(x,ds,a), echo(x,ds,a), ds)[]
move?x:pnt;  sldrADU[...](input(x,ds,a), echo(x,ds,a), ds) []
release?x:pnt;  sldrADU[...](input(x,ds,a), echo(x,ds,a), ds)[]
send!result(a);  sldrADU[...](a, dc, ds) []
output!dc;  sldrADU[...](a, dc, dc) []
fromContainer?x:rct; sldrADU[...](receive(a,x),render(dc,x), ds)
endproc
```

The CU for the interactor *sldr* is process *sldrCU* defined below by the parallel composition of partial constraints of its behaviour. These constraints are defined as distinct processes which describe the sequencing of interactions which effect clicking, dragging, input and instant feedback respectively:

```
process sldrCU[press, move, release, output, fromContainer, send] : noexit :=
```

```
  ((click[press, release] ||| drag[press, move, release])
          |[move, release]|
  lazyInput[move, release, send])
          |[press, move, release, fromContainer, send]|
  instantFeedback[press, move, release, output, fromContainer, send]
endproc
```

Note the use of interleaving and partial synchronisation in the last example. Processes *click* and *drag* are combined with the interleaving operator |||. This means that the corresponding temporal orderings are related by logical disjunction: a slider is either clicked or dragged. Interactions *move* and *release* require the synchronisation with process *lazyInput*. This corresponds to a logical conjunction of the temporal constraints for these interactions. The consistent use of this technique, is called the *constraint oriented specification style* (Vissers, Scollo, van Sinderen, Brinksma 1991) which affords increased modularity and re-usability of the specification components.

The following process definitions specify some of the individual constraints combined by *sldrCU*:

```
process click[press, release]: noexit:=
 press?x:pnt; release?x:pnt; click[press, release]
endproc

process drag[press, move, release] : noexit :=
 press?x:pnt; (repeat[move] [> release?x:pnt; drag[press, move, release])
endproc

process repeat[move]: noexit :=
 move?x:pnt; repeat[move]
endproc

process lazyInput[move, release, send]:noexit:=
 move?x:pnt; send?x:int; lazyInput[move, release, send] []
 release?x:pnt; send?x:int; lazyInput[move, release, send]
endproc
```

Process *click* simply is defined as a press-release event-sequence. Process *drag* specifies that after an interaction *press*, repeated mouse movement is possible, which is interrupted by an interaction *release*. The interruption of *repeat[move]* is specified with the disable operator [>. Markopoulos (1997) identifies a range of such interactive behaviours, such as continuous feedback, toggle, trigger, etc, which are specified in a similar fashion, and which can be used as constituents of different interactors.

8 Composition and Synthesis of Interactor Specifications

Complex interface specifications can be constructed by composing ADC interactors hierarchically. The binary composition of interactors is specified in LOTOS by a behaviour expression of the form

$$DF := ADC_A \otimes ADC_B$$

where $\otimes$ is one of the LOTOS operators ||, |||, [>, [] and |[G]|.

The LOTOS operator |[G]| specifies the partial synchronisation of ADC_A and ADC_B over a gate-set $G \subseteq G_A \cup G_B$. Full synchronisation || and pure interleaving ||| are boundary cases of the partial synchronisation of behaviour expressions, where $G=G_A \cup G_B$ and $G=\emptyset$ respectively.

- The synchronisation over g can specify data flow from one interactor to another, synchronised input from a third interactor, conjunction of the temporal ordering constraints they each specify in their controller, etc. (see Markopoulos, Rowson and Johnson 1997).
- The choice operator [] can be used to specify alternative interactions. Consider, for example, a set of interactors for drawing different shapes on a drawing package invoked by some interactor supporting logical disjunction, e.g., a ‘palette’ or as ‘radio buttons’. The alternative interactors can be related by the choice operator and their composition can be synchronised with the ‘palette’ interactor for their initialisation.
- The disable operator [> can help specify interruption. For example, the interruption of a task supported by an interactor, e.g., a dialogue box, can be modelled by composing this interactor with one modelling an ‘ok’ or a ‘cancel’ button using the disable operator.

EXAMPLE

Consider the list interactor of figure 6 specified as the composition of its ADU and its CU. (Suppose L, SL are initial values for the state parameters of the ADU):

```
process lst[dinp, dout, ainp, aout] : noexit :=
 lstADU[dinp, dout, ainp, aout](L, SL, SL)
  |[dinp, dout, ainp, aout]|
 lstCU[dinp, dout, ainp, aout]
endproc
```

Process *lstADU* follows from the general definition of ADU.

```
process lstADU[dinp, dout, ainp, aout](a: lstel, dc, ds: scrLst) : noexit :=
 aout!result(a);lstADU[dinp, dout, ainp, aout](a, dc, ds) []
 dout!dc; lstADU[dinp, dout, ainp, aout](a, dc, dc) []
 ainp?x:lstel; lstADU[dinp, dout, ainp, aout](receive(a,x), render(dc,x), ds)
[]
 dinp?x:pnt; lstADU[dinp, dout, ainp, aout](input(x,ds,a), echo(x,ds,a), ds)[]
 dinp?x:Int; lstADU[dinp, dout, ainp, aout](input(x,ds,a), echo(x,ds,a), ds)
endproc
```

The only constraint specified by *lstCU* is that the interactor give immediate feedback for input at *ainp* and *dinp* gates.

```
process lstCU[dinp, dout, ainp, aout]: noexit :=
 ainp?X:lstel; dout?X:scrLst; lstCU[dinp, dout, ainp, aout] []
 aout?X:el;    lstCU[dinp, dout, ainp, aout] []
 dout?X:scrLst;   lstCU[dinp, dout, ainp, aout] []
 dinp?X:Int; dout?X:scrLst; lstCU[dinp, dout, ainp, aout]
endproc
```

The composition of the list and the slider interactor is specified as follows (where formal gate parameters have been actualised).

```
sldr[press, move, release, output, fromContainer, send]
 |[send, fromContainer]|
lst[click, output, fromContainer, send]
```

Note that both interactors may receive input from their container synchronously, while *sldr* sends its result to *lst* via gate *send*.

SYNTHESIS OF ADC INTERACTORS

LOTOS behaviour expressions specifying the composition of interactors, as discussed in the previous paragraph, can be operands in more complex behaviour expressions. It is desirable that the resulting expressions can be interpreted as ADC interactors. This is useful for determining the role of each gate, the type of connections, so as to instantiate generic constraints and expressions of interaction requirements. For example, is gate *send* a graphical input gate (role *dinp*), or does it become an *aout* gate? Also, is *fromContainer* an *ainp* gate for the composite interactor? Markopoulos, Rowson and Johnson (1997) define the formal transformations of composition expressions to ADC interactor specifications. The relevant discussion can be summarised in the form of the following theorems.

Theorem 1. Synthesis of synchronous composition expressions.

A behaviour expression which specifies the synchronous composition of two ADC interactor specifications ADC^A and ADC^B over a gate-set G, such that $G \cap G_o^A \cap G_o^B = \varnothing$, can be rewritten as a strongly equivalent ADC interactor. The CU of the resulting interactor is formed by the parallel composition of the two component CUs over the gate-set G and its ADU is formed by the parallel composition of the component ADUs over a gate-set:

$$G_1 = G_{io}^A \cap G_{io}^B \cap G$$

where the gate-sets for interactors ADC^A and ADC^B are indicated accordingly (figure 8). This rewriting, termed synthesis, preserves strong bisimulation equivalence (Milner 1989), denoted by ~.

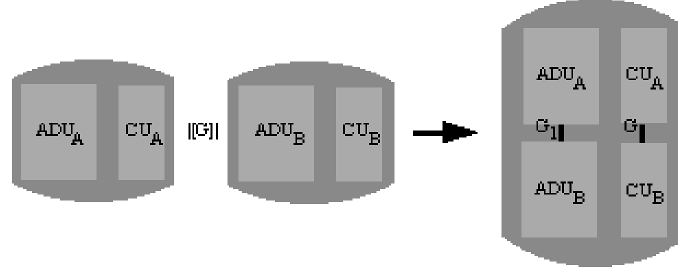

Figure 8. Synthesis applied to the synchronous composition of two interactors

$$(ADU_A|[G^A_{io}]|CU_A)|[G]|(ADU_B|[G^B_{io}]|CU_B) \sim (ADU_A|[G_1]|ADU_B)|[G^A_{io} \cup G^B_{io}]|(CU_A|[G]|CU_B)$$

Because of condition $G \cap G^A_o \cap G^B_o = \emptyset$ synthesis does not apply to interactors which synchronise on their output gates.

The right-hand side of the above equivalence specifies an ADC interactor for which the roles of its gate sets are re-defined as follows:

$$G^{AB}_{dout} = G^A_{dout} \cup G^B_{dout},\ G^{AB}_{aout} = G^A_{aout} \cup G^B_{aout},\ G^{AB}_o = G^A_o \cup G^B_o$$

$$G^{AB}_{dinp} = G^A_{dinp} \cup G^B_{dinp} - G^{AB}_o,\quad G^{AB}_{ainp} = G^A_{ainp} \cup G^B_{ainp} - (G^{AB}_o \cup G^{AB}_{dinp})$$

$$G^{AB}_c = (G^A_c \cup G^B_c) - (G^{AB}_{dinp} \cup G^{AB}_{ainp} \cup G^{AB}_{dout} \cup G^{AB}_{aout})$$

■

By construction G_1 ensures that no synchronisation between ADU and CU is introduced by the rewriting. This theorem can be simplified for the case of interleaving where $G = \emptyset$:

$$(ADU_A|[G^A_{io}]|CU_A)|||(ADU_B|[G^B_{io}]|CU_B) \sim (ADU_A|||ADU_B)|[G^A_{io} \cup G^B_{io}]|(CU_A|||CU_B)$$

In the case of full synchronisation, where $G = (G^A_{io} \cup G^A_c) \cup (G^B_{io} \cup G^B_c)$ and provided that $G \cap G^A_o \cap G^B_o = \emptyset$, the equivalence may be simplified to the following form:

$$(ADU_A|[G^A_{io}]|CU_A)||(ADU_B|[G^B_{io}]|CU_B) \sim (ADU_A|[G^A_{io} \cap G^B_{io}]|ADU_B)|[G^A_{io} \cup G^B_{io}]|(CU_A||CU_B)$$

Theorem 2. Synthesis of dynamic composition expressions.

The composition of two ADC interactor specifications with a dynamic composition operator, i.e. choice [] or disable [>, can be rewritten as a strongly equivalent ADC interactor. The CU of this interactor is a behaviour expression which combines the two component CUs with the same dynamic operator. The ADU of the compound interactor is formed by the interleaved composition of the two component ADUs. The roles of the gates are preserved in the compound interactor:

$$(ADU_A|[G^A_{io}]|CU_A) \otimes (ADU_B|[G^B_{io}]|CU_B) \sim (ADU_A|||ADU_B)|[G^A_{io} \cup G^B_{io}]|(CU_A \otimes CU_B)$$

where $\otimes$ is either [] or [> and

$$G^A_c \cap G^B_{io} = \emptyset \quad \wedge \quad G^B_c \cap G^A_{io} = \emptyset \quad \wedge \quad G^A_{io} \cap G^B_{io} = \emptyset$$

■

For example, consider the scrollable list which was specified by the synchronous composition of the interactors *lst* and *sldr*. By the synthesis transformation this composition can be rewritten as a single interactor whose ADU and CU are as follows:

```
adu[press, move, release, output, fromContainer, send, click](BV, SB, L, SL)
        |[press, move, release, output, fromContainer, send, click]|
cu[press, move, release, output, fromContainer, send, click]
```

where the ADU for this interactor is defined as

```
process adu[...](BV:boundValue, SB:sliderBar, L:lstel, SL:scrLst):noexit:=
  sldrADU[press, move, release, output, fromContainer, send](BV, SB, SB)
        |[send, fromContainer]|
  lstADU[click, output, fromContainer, send](L, SL, SL)
endproc
```

Notation	Meaning
$q \xrightarrow{\mu_1..\mu_n} r$	$\exists q_1...q_n \mid q \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} q_2 \to ... \to q_{n-1} \xrightarrow{\mu_n} r$
$q \Rightarrow^{\varepsilon} r$	$q \equiv r \vee \exists n \geq 1 \mid q \xrightarrow{\tau^n} r$
$q \Rightarrow^{\mu} r$	$\exists q_1, q_2 \mid q \Rightarrow^{\varepsilon} q_1 \xrightarrow{\mu} q_2 \Rightarrow^{\varepsilon} r$
$q \Rightarrow^{\mu_1..\mu_n} r$	$\exists q_1..q_{n-1} \mid q \Rightarrow^{\mu_1} q_1 \Rightarrow^{\mu_2} ... \Rightarrow^{\mu_{n-1}} q_{n-1} \Rightarrow^{\mu_n} r$
$q \not\Rightarrow^{\mu_1..\mu_n}$	$\nexists r \mid q \Rightarrow^{\mu_1..\mu_n} r$
Tr(q)	$\{\sigma \in A^* \mid \exists r \bullet q \Rightarrow^{\sigma} r\}$

Table 1. Some notation to describe traces of processes. Traces are sequences of observable actions in a set A that describe transitions from the initial state q of a process to some other state r. Internal (non-observable) actions are denoted as τ.

and the controller is defined as

```
process cu[...]:noexit:=
  sldrCU[press, move, release, output, fromContainer, send]
         |[send, fromContainer]|
  lstCU[click, output, fromContainer, send]
endproc
```

9 Formalising abstract interaction properties with the ADC model

Following the discussion on abstract models, in section 3, the formal expression of properties like predictability and observability requires the formalisation of two notions:

- the instantaneous 'state' reflecting the internal workings of an interactor, and
- the distinct 'views' of an interactive system reflecting its observation through a display.

In the process algebraic framework of LOTOS, the notion of 'state' corresponds to a behaviour expression that describes the future behaviour of a process. Comparing states amounts to comparing behaviour expressions, for example, they may be strong observational equivalent, weak observational equivalent, etc., (see, e.g., Milner 1989, de Nicola 1989).

Let P denote a behaviour expression describing an interactor. P may offer to its environment a set of interactions, denoted as *out(P).* Let $out_G(P)$ denote the interactions offered on a set of gates G:

$$out_G(P) = \{g \in G \mid P \Rightarrow^{g}\} \cup \{g < v > \mid g \in G \wedge v \in valueSet(g) \wedge P \Rightarrow^{g<v>}\}$$

where the relation $\Rightarrow^g$ denotes that there is an observable transition labelled by g from P to the behaviour described by the right hand side of the relation (see table 1); *valueSet(g)* denotes the set of all possible values that may be associated with interactions on gate g. For an input of data specified as *g?x:s,* where s is some sort identifier, *valueSet(g)* includes all possible values of this sort. For a simple output of data g!v, *valueSet(g)* contains the value v.

Similarity of Interactors. An interactor may be associated with multiple *result* operations. For each gate $g \in G_{aout}$, $out_G(P)$ contains a single interaction *g!result(A)*, where A is the abstraction held by the interactor. For each gate $g \in G_{dout}$, $out_G(P)$ contains a single interaction *dout!D*.

Two interactors P and Q (or two states of the same interactor) are called *abstraction-similar* (*display-similar*) when they are defined with identical gate-sets G_{aout} (respectively G_{dout}) and when they output the same values on those. To exclude the contrived case where no events are offered on a gate, the clause is added that these sets should not be empty.

$$similar_A(P,Q) \text{ iff } out_{G_{aout}}(P) = out_{G_{aout}}(Q) \neq \emptyset$$
$$similar_D(P,Q) \text{ iff } out_{G_{dout}}(P) = out_{G_{dout}}(Q) \neq \emptyset$$

Abstraction-side and Display-side behaviours. Interactors can be 'observed' from the abstraction or the display sides. The *abstraction-side* behaviour of the interactor is observed by interaction at gates

Display Predictability	$similar_D(P,Q) \Rightarrow D_P \approx D_Q$
Result Predictability	$similar_A(P,Q) \Rightarrow A_P \approx A_Q$
Honesty	$similar_D(P,Q) \Leftrightarrow similar_A(P,Q)$
Trustworthiness	$D_P \approx D_Q \Leftrightarrow similar_A(P,Q)$
Strong WYSIWYG	$similar_D(P,Q) \Leftrightarrow A_P \approx A_Q$
Weak WYSIWYG	$D_p \approx D_Q \Leftrightarrow A_P \approx A_Q$

Table 2. Expressions of predictability and observability properties.

G_{aout} and G_{ainp}. The *display-side* behaviour is observed at gates G_{dinp} and G_{dout}. They are defined as follows:

$$A_P = hide\,(G_c \cup G_{dinp} \cup G_{dout})\; in\; \mathrm{P}$$

$$D_P = hide\,(G_c \cup G_{ainp} \cup G_{aout})\; in\; \mathrm{P}$$

Abstraction and Display Equivalence. Two interactors P and Q (or two states of the same interactor) are called *abstraction equivalent* if their abstraction-side behaviours are observationally equivalent, i.e. $A_P \approx A_Q$. They will be called *display equivalent* when their display-side behaviours are observationally equivalent, i.e. $D_P \approx D_Q$.

The meaning of these definitions depends on the equivalence relation $\approx$ they stipulate. The choice between equivalence relations for processes is a contentious issue, cf. de Nicola (1989), which, in the present context, depends on the ability of humans to tell apart interactive behaviours and to detect and interpret differences of the display contents. Without wishing to postulate a theory of user perception or cognition, weak observational equivalence is used here as it distinguishes processes only with respect to observable interactions with their environment and its verification is supported by model checking tools.

Table 1 summarises the definitions of observability and predictability properties. Consider, for example, the first row of table 2. An interactor is called *display predictable*, if the similarity of two instances of its display implies that they are also display equivalent. In other words, the display status of the interactor determines its display-side behaviour. A symmetrical definition of *result predictability* can be written as in row 2 of table 2.

The last four rows of the table describe observability-related properties as logical equivalences $\Leftrightarrow$. The two directions of implication correspond to a static (*if*) and a dynamic (*only if*) aspect of each property. For example, consider an electronic messaging application that operates in the background, while the user is engaged in unrelated interactions. The user is alerted to the arrival of a new message by an icon appearing on the display. Consider also a user who from a change of the display infers that the state of the messaging application has changed. When the display has not changed, i.e. there is no icon on the screen, the user infers that no new message has arrived. A system for which the implication $similar_D(P,Q) \Rightarrow similar_A(P,Q)$ holds warrants the second inference but not the first, so it is *statically honest*. A system that displays a message-arrival icon without reason is still statically honest. An interface that satisfies the dynamic honesty requirement $similar_A(P,Q) \Rightarrow similar_D(P,Q)$ will correct this problem. On the other hand, a system that does not display the icon when a message arrives is dynamically honest but not statically honest. True honesty should require both conditions to hold.

Unfortunately, these definitions are too strong. A closer look shows that no reasonable ADC interactor can be display predictable. By the definition of the model, immediately after any input action on the display side, similarity is maintained:

$$\forall P \in S \mid g < v > \in out_{dinp}(P) \wedge P \xrightarrow{g<v>} Q \bullet similar_D(P,Q)$$

However, after an output event the display behaviour changes. According to the definition of Table 1, for the interactor to be predictable we can infer that the display side behaviour should not change:

$$\forall P \in S \mid g < v > \in out_{dinp}(P) \wedge P \xrightarrow{g<v>} Q \bullet D_P \approx D_Q$$

Clearly, this cannot be true for any useful interactive system, as user interactions do not have any impact on the display. This problem arises because we reason about the state of the interactor using an event-based model. The required relationship between the display status and its future behaviour breaks down between successive input and output events. To get around the problem, the similarity predicate should only apply to a subset of the states of the interactor, say, to states that may precede an input. This will ensure that the relevant properties hold whenever the user is able to influence the behaviour of an interactor. Therefore the similarity predicate is redefined:

$$similar_A(P,Q) \; iff \; out_{aout}(P) = out_{aout}(Q) \neq \emptyset \wedge out_{dinp}(P) \neq \emptyset \wedge out_{dinp}(Q) \neq \emptyset$$

$$similar_D(P,Q) \; iff \; out_{dout}(P) = out_{dout}(Q) \neq \emptyset \wedge out_{dinp}(P) \neq \emptyset \wedge out_{dinp}(Q) \neq \emptyset$$

Unfortunately, the practical utility of these formal expressions is limited. Verification by hand or by model checking of these properties depends very much on the judicious choice of abstraction level and on crafting the data types used so that they produce finite sets of values. This can be quite hard if the specification addresses non trivial interaction. It is easier to detect when these properties do not hold (see Markopoulos 1997).

10 Dialogue verification with ADC

By the definition of the ADC model, dialogue, i.e. the temporal ordering of interactions with the user, is solely and exclusively modelled within the CU. The roles of the gates classify interaction events, and this enables the expression of generic dialogue properties.

An example of a dialogue property is *eventual feedback*: A user input action shall eventually generate some feedback. Using an action based temporal logic called ACTL (Bouali, Gnesi and Larosa, 1994) we can express this with the following predicate:

$$\mathrm{AG}[dinp]\ \mathrm{E}(tt\ {}_{tt}U_{dout}\ tt)$$

This expression can be interpreted as follows. Always (operator AG), from the state following a user input on a graphical input gate *dinp*, there exists (operator E) a sequence of events which lead up to an event on a gate *dout*. This formula can be instantiated for a given interactor specification, e.g., for the slider interactor of section 7 it will be written as:

$$\mathrm{AG}[press]\ \mathrm{E}(tt\ {}_{tt}U_{output} tt)$$

Another example is that the press of a button will not be ignored, i.e. an appropriate event *notify* will be sent to the application

$$AG[press]\ E(tt\ {}_{tt}U_{notify} tt).$$

Model checking tools support the verification of such properties (e.g. Bouali et al. 1994). A practical alternative for dialogue verification is to specify dialogue properties as LOTOS processes. These processes are identical to those used inside the CU of the interactor, so the same dialogue specifications can be used for both purposes. For example, the LOTOS process below specifies that an occurrence of an input on the display side is followed by the occurrence of an output before another input is allowed. An output event is still allowed to happen without a prior input event.

```
process feedback[dinp, dout] : noexit :=
  dinp; dout; feedback[dinp, dout]  []
  dout; feedback[dinp, dout]
endproc
```

To verify that for our slider a *press* event must be followed eventually by an output event, the interactor *sldrCU* is compared to process *feedback*. For this comparison all actions that are not input or output on the display side are 'hidden', since they do not concern this property. The *sldrCU* interactor can then be compared to the feedback property, using a model checking tool such as CADP (Fernadez, et al. 1992) to verify the following weak observational equivalence:

```
hide (G - {press, output}) in P(sldrCU) ≈ feedback[press,output]
```

We verify weak observational equivalence, because we wish to ignore internal actions of the interactor and the actions hidden in the last expression. Note that, this equivalence describes the same dialogue property as the ACTL definition of eventual feedback.

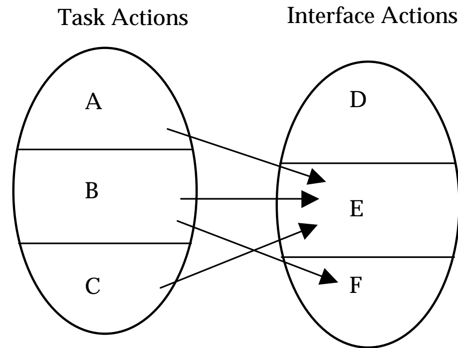

Figure 9. Correspondence of task and interface actions. Arrows indicate potential mappings between task and interface actions.

In conclusion, the ADC interactor provides a conceptual and formal framework for specifying and verifying dialogue properties. Interactors prompt the expression of dialogue properties in terms of architectural constructs, here the gates of the interactors and their respective roles. Note also that by repeated application of theorems 1 and 2, complex configurations of interactors specifying an interactive system can be transformed to a single interactor. Because synthesis redefines the role of each gate the formal specification of dialogue properties is the same for a compound interactor as it is for a simple interactor.

FORMALISING THE CORRESPONDENCE OF TASK AND INTERFACE ACTION SEQUENCING

An established tenet in the field of human-computer-interaction is that the design of user interfaces should be based on models of users' task knowledge. Such a design approach is called *task-based*. Examples of task based design approaches are ADEPT (Wilson and Johnson 1996) and MUSE (Lim and Long 1994). Wilson and Johnson (1996) describe two principles for designing the dialogue of the user interface:

- Task actions, which are considered as elementary components of task activity, should be mapped to elementary interactions with the system.
- The interface dialogue should not violate task sequencing knowledge, i.e. it should not force the users to perform their tasks in a different order than that of the task model.

Omitting the details of specifying task models (see for example Markopoulos, Wilson and Johnson 1994) it is assumed here that both the task model (TM) and the interface model (IM) are specified as LOTOS processes. First, we discuss the implications of the above two principles for the set of task actions TA specified in TM and the set of interface actions (IA) specified by IM. We can partition these sets to the following subsets (see figure 9):

A: Task actions representing task activity which is not input to the system.

B: Task actions which are directly supported by the system.

C: Task actions which concern the operation of the interface rather than the task itself. Whether set C is empty or not depends on the modelling approach taken: does the task describe how to operate the interface or does it abstract away from system related issues?

D: Interface actions which are not observable to the user. They represent non-determinism resulting from internal system behaviour, e.g., communication between different system sub-components.

E: Interface actions directly relevant to the performance of the task in question.

F: Interface actions observable to the user but which do not concern the specific task.

The constitution of TA and IA embodies significant modelling choices. Actions in set A may model aspects of cognitive activity, e.g., decisions, recall, etc., or externalised behaviour of the user which the system has no way of perceiving. Modelling these aspects of interaction can be approached from a range of perspectives which are not excluded by the simple model discussed here. While actions in sets A and D are important elements of the task and interface models respectively, in modelling their

interaction these two subsets must be considered internal (hidden) transitions. This means that they are observed as non deterministic behaviour.

Actions are particular to an abstraction level for the description of the tasks and interfaces. In task based design this level of abstraction is determined by a task analysis (Johnson 1992). For example, consider the task of editing a graph using a simple drawing tool. The graph is constructed by connecting lines and nodes. The tool is general purpose, so when moving a node, all the lines connected to the node have to be adjusted manually. This suggests a mismatch between the levels of abstraction of the task and that supported by the interface. A purpose-specific tool that supports the task of graph editing will allow the user to move the node with its edges following.

The task model may include elements of task knowledge which are specific to the manipulation of the interface and which enable the users to perform their tasks (then $C \neq \emptyset$). Alternatively, TM can describe idealised task knowledge which does not describe such aspects of the task (then $C=\emptyset$). In such a case the corresponding interface actions should also be abstracted away from. The comparison with the task model will reflect only on the feasibility of performing the idealised task, irrespectively of how complicated the interaction may be. A more accurate model of interaction results if task actions C are explicitly mapped to interface actions E. Including such task actions and their corresponding interaction in the formal representations gives a better handle for comparing the two and a stronger task requirement.

Consider now the binary relationship R between task actions and interface actions:

$$R = TA \leftrightarrow IA$$

R models a competent user who can relate task actions to the interactions supported by the interface. This competence can not always be assumed, it is however desirable to support it by design. The properties of R itself can give interesting insights into the properties of the designed interface.

- Suppose $R: B \cup C \rightarrow E \cup F$ is a function. This would mean that the interface supports all task actions.
- Suppose $R: B \cup C \rightarrow E \cup F$ is surjective (or according to the definitions above $F=\emptyset$). Then, the interface does not support task actions outside those of the task model.
- Suppose $R: B \cup C \rightarrow E \cup F$ is injective. This implies that there is no redundancy; each task action is supported by distinct interface actions.

For example, consider the task of formatting a document on a word processor. R is a function if all actions for the formatting task are supported by the system. Normally, R is not surjective, as there will be functionality that the user is not aware of or has not yet discovered. R is normally not injective since there are multiple ways of achieving each task action. For example, setting the margins of a document can be done by a dialogue box or by direct manipulation of the on-screen representation of borders.

Specifying the relation R. R can be specified as a two-step renaming of TA and IA to a set of labels L as follows:

$$TM_H = TM[H(TA)] \text{ where } H: TA \rightarrow L$$

$$IM_G = IM[G(IA)] \text{ where } G: IA \rightarrow L$$

$$\mathrm{H(B \cup C) = G(E), H(A) = \{x\}, G(D) = \emptyset}, G(F) = F$$

The correspondence between task and interface actions is specified in two stages as $R=H;G^{-1}$ where the symbol ; denotes the composition of binary relations. Combined, the two mappings of labels from TA to a set of labels L and then to a set of labels IA have the effect of matching task actions to interface actions. All actions in set A are mapped to a reserved label x, so that they can be treated uniformly in further analysis.

The preservation of task action sequencing in the interaction dialogue can be modelled formally in terms of the conformance relation between processes, introduced by Brinksma (1989).

The Confomance Relationship. Let Q_1 and Q_2 be two processes and let L be the set of all possible actions for all processes.

Q_1 conf Q_2 if

$$\forall \sigma \in Tr(Q_2) \wedge \forall A \subseteq \mathsf{L} \bullet if\ \exists Q_1' \mid \forall \alpha \in A \bullet Q_1 \Rightarrow^{\sigma} Q_1' \not\Rightarrow^{\alpha} \text{ then } \exists Q_2' \mid \forall \alpha \in A \bullet Q_2 \Rightarrow^{\sigma} Q_2' \not\Rightarrow^{\alpha}$$

This expression is read as follows: If Q_1 can perform some trace σ, then behave like a process Q_1', and if Q_2 can perform the same trace σ and then behave like Q_2', then the following conditions are required: whenever Q_1' refuses to perform an action α from a set A then Q_2' must also refuse every action in A.

The requirement not to violate task sequencing, can now be formulated as follows: An interface that behaves as IM, will not reach an impasse when performing a task, as described in TM, when it is possible to support a mapping of task actions A to interactions on gates G of the interface, as defined by R so that: IM_G conf TM_R.

Conformance can be tested. This means that a set of tests can be defined which can assess whether an implementation (here the interface model) conforms to a given specification (here the task model). Software tools support the generation and execution of tests from a specification (Mañas 1992).

EXAMPLE

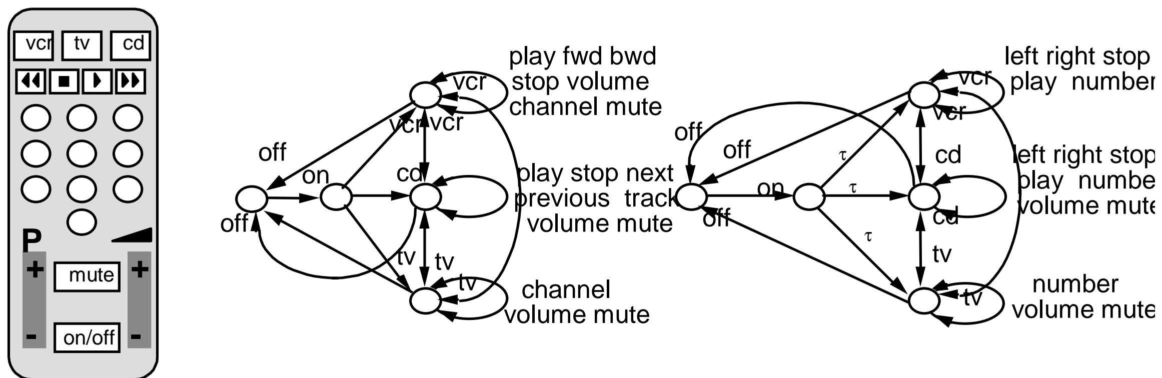

Figure 4. An 'all-in one' remote control for a VCR, a CD and a TV.

Figure 10 illustrates a remote control for the combined operation of three systems: the TV, the VCR and a CD player. For the sake of the example, let the labelled transition system to the left represent the task model of operating the system and the one to the right the interface model. We note that:

- In this task model the user selects one of these systems directly after switching the system on. Further, the user expects to change the volume or to 'mute' the output while operating any of the devices.
- When switched on, the system will non-deterministically choose one device to operate, e.g., the last device that was operated before switching it off. Also, the system is 'moded'. The remote control does not allow manipulation of the volume or muting when in the VCR 'mode'. Rather the user must switch to the TV to adjust/mute the volume.

This example illustrates how different task actions are mapped to the same interface action. For example, moving the video tape forward and moving to the next track on a CD are supported by the same interface action. The numeric pad may support choosing a track on the CD and also selecting the channel on the TV or the VCR. The sets of actions of figure 9 will now be as follows, with the obvious mappings between B and E:

- A=∅. So other activities of the user or how they decide which device to switch on are not modelled in the task.
- B={on, off, vcr, play, fwd, bwd, stop, volume, channel, mute, next, previous, track}.
- C=∅. A very good fit of the interface to the task is assumed here. A counter example would be if there were extra operations, say to switch on the remote controller separately from the devices.
- D=∅. The internal workings of the system are not modelled, e.g., the communication with the devices, or how it determines which one to activate after it is switched on.
- E={on, off, left, right, stop, play, number, volume, mute}.
- F=∅. The interface does not support any more tasks than those included in the task model.

For these two systems IM does not conform to TM. This can be established by testing the interface against the following test (the operator ; denotes action succession).

t = on; vcr; mute; success; stop

This test must fail (reach an impasse). However, if the system was modified so that *mute* and *volume* are offered in all modes we would have successful verdict. IM would then conform to TM. They would still not be equivalent, since when the system is switched on the TM does not assume that a device is selected by default.

Establishing a correspondence between tasks and interface specifications is a long standing aim of human-computer interaction research (see for example Johnson 1992). Few attempts have been made to define this relationship explicitly and at a level of abstraction that describes tasks macroscopically rather than as low level interaction tasks. The conformance relationship discussed here concerns not only the feasibility of performing a task but also the preservation of task sequencing for that task.

11 Discussion

An important motivation for the research reported has been to facilitate the formal specification and verification of interactive systems. A key to achieve this aim is the definition of appropriate abstractions. This paper has argued that interactors are appropriate intermediate-level abstractions which help specify generic and re-usable formalisations of user interface software. The ADC interactor model supports the formal specification of user interfaces by providing a standard mapping of user interface software architecture constructs to LOTOS language elements and a standard style of writing specifications, thus relieving the specifier from decisions which are costly to resolve on a problem by problem basis. For the field of formal specification, this idea of 'specification styles' is analogous to the popular concept of 'design patterns' for programming.

An important consideration for this work has been to use a standard specification language with a wide user base, rich documentation and tool support. The formalisation of the ADC model describes a regular style of writing and composing specifications in LOTOS, rather than a syntactic or semantic extension of the language which would not be supported by off-the-shelf tool support. Interactors are specified by instantiating the ADC interactor model and the individually required adaptations are restricted to localised parts of the interactor specification:

- the equations component of the data type,
- the problem-specific data types composed by the data type AD (like data type *boundedVal* and *sliderBar* of the slider-example), and
- the constraints used within the CU.

The mapping between the dynamic component of the specification and the data specification component is described by the ADU in a fixed way. Given the sorts of data which are communicated over the gates of the ADU and a characterisation of the role of each gate, the formal specification of the ADU follows mechanically. While this does not reduce the expressiveness of the specification language, it results in a consistent specification style which is easier to write and to understand than unstructured LOTOS.

The reuse of formal specifications can be supported at various levels:

- individual interactors or composite interactors can be re-used in different design contexts,
- data type specifications and dialogue constraints can be re-used for the definition of new interactors.

The formal specification of the ADC interactor inherits both advantages and limitations of LOTOS. Specification and verification are supported by widely used general-purpose tools which are evolving as a result of continuing research in formal methods. LOTOS is particularly powerful in specifying the dynamic behaviour of interactive systems, but is not equally adept at specifying state invariants or display-layout properties. It can model the lifetime of interactors for which the connections and the context of execution are specified statically in advance. However, the approach discussed does not generalise easily to an indefinite number of interactors, created and destroyed dynamically or whose configuration changes dynamically.

The ADC interactor model borrows concepts originating from user interface software architectures. Grounding a formal model on software architectures results in representations relevant to the

concerns of a software developer which encapsulate design in re-usable specification components. For validation purposes, the ADC model has been used as an implementation architecture as well. The mapping to programming language elements was straightforward, but the formal specification of software architecture is excessively complex to be used as an implementation blue-print in practice.

Markopoulos, Papatzanis and Johnson (1998) report a case study which uses the ADC interactor model as a conceptual framework for the informal specification of user interfaces. This informal specification is seen as an intermediate step towards a formal specification, but also, as a useful notation in its own right, which does not incur prohibitive formalisation costs. The case study compares the architectural description of user interface designs using the ADC model to a task-based description in UAN (Hix and Hartson 1993), with respect to the ability of the two notations to capture design recommendations that result from a usability evaluation. The case study results are very encouraging with respect to the coverage of design issues by the architectural description.

A more practical approach for designing and specifying user interface architectures is to use component interconnection notations. Contrary to interactor models, these formalisms detail the software structure and the methods that software components offer or require from the context of deployment, but they do not model their behaviour. Markopoulos, Shrubsole and de Vet (1999) use such a formalism to describe PAC-like interactors. Component interconnection notations do not offer the analytical power of interactors but are more easily used during software development.

An important issue flagged in this paper concerns the validity of the formal verification. Analytical results from earlier formal models of user interface software, both abstract and concrete, have been reproduced in the framework of the ADC model. It has been mentioned already that a theoretically founded formalisation of usability related properties cannot be derived from a model of the system only. Rather, system models must be related to models of user tasks, behaviour, domain models, etc. The discussion on task conformance is an example, albeit limited to action sequencing, of how a formal framework can help relate diverse modelling approaches with the aid of some clearly defined principles for relating the two modelling domain.

12 References

BOLOGNESI, T. & BRINKSMA, E. (1989). Introduction to the ISO specification language Lotos, in VAN EIJK, P., VISSERS,C. & DIAZ, M., Eds., *The Formal Description Technique Lotos*, Elsevier, North-Holland, 23-73.

BOUALI, A., GNESI, S. & LAROSA, S. (1994). The Integration Project for the JACK Environment. *Bulletin of the EATCS*, 54, 307-223.

BRINKSMA, E. (1989). A theory for the derivation of tests. In VAN EIJK, PHJ., VISSERS, C.A. & DIAZ, M., Eds., *The Formal Description Technique Lotos*, Elsevier, North-Holland, 235-247.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., STAL, M. (1996). A system of patterns. John Wiley & Sons, Chichester.

COUTAZ, J. (1987). PAC, an Object Oriented Model for Dialog Design. In BULLINGER, H.J., SHAKIEL, B., Eds., *INTERACT '87*, Elsevier, North-Holland, 431-436.

COUTAZ, J. (1997). PAC-ing the User Interface Architecture. In HARRISON, M.G. & TORRES, J-C, Eds., *Design, Specification and Verification of Interactive Systems'97,* Springer-Wien, 13-28.

DE NICOLA, R. (1989). Extensional Equivalences for Transition Systems, *Acta Informatica*, 24, 211-237.

DIX, A.J. (1991). *Formal Methods for Interactive Systems,* Academic Press.

DUKE, D.J. & HARRISON, M.D. (1994). From Formal Models to Formal Methods. In TAYLOR RN & COUTAZ J, EDS., *Software Engineering and Human-Computer Interaction*, ICSE'94 Workshop on Software Engineering and Human Computer Interaction, Springer-Verlag, LNCS 896, 159-173.

FACONTI, G.P. (1993). *Towards the Concept of Interactor*, AMODEUS project report, ref. sm/wp8.

FERNANDEZ, J.C., CARAVEL, H., MOUNIER, L., RASSE, A., RODRÍGUEZ, C. & SIFAKIS, J. (1992). A toolbox for the verification of LOTOS Programs. *ICSE '92, 14th Int. Conference on Software Engineering*, Melbourne, May 1992.

HARRISON, M.D. & DIX, A.J. (1990). A state model of direct manipulation in interactive systems. In HARRISON, M.D. & THIMBLEBY, H.W., Eds., *Formal Methods in Human Computer Interaction*, Cambridge University Press, 29-151.

HIX, D. & HARTSON, R.H. (1989). *Developing User Interfaces: Ensuring Usability Through Product and Processes*. Wiley, New-York.

ISO (1989). *Information Processing Systems-Open Systems Interconnection-LOTOS-A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807, International Organisation for Standardisation, Geneva.

JOHNSON, P. (1992). *Human-Computer Interaction: Psychology, Task Analysis and Software Engineering,* McGraw-Hill, London, 1992.

KRASNER, G.E. & POPE, S.T. (1988). A Cookbook For Using the Model-View-Controller User Interface Paradigm in The Smalltalk-80 System. *Journal of Object Oriented Programming*, 1, 3,26-49.

LIM, K.Y. & LONG, J. (1994) *The MUSE method for usability engineering*, Cambridge University Press.

MAÑAS, J.A. (1995) Getting to use the LOTOSPHERE integrate tool environment (LITE). In BOLOGNESI, T. & VAN DE LAGEMAAT, J., (1995) *LOTOSphere: Software Development with LOTOS*, Kluwer (Netherlands).

MARKOPOULOS, P. (1997). *A compositional model for the formal specification of user interface software*, Ph.D. Thesis, Department of Computer Science, Queen Mary and Westfield College, University of London, March 1997.

MARKOPOULOS, P., PAPATZANIS, G., JOHNSON, P. & ROWSON, J. (1998). Validating semi-formal specifications of interactors as design representations. In MARKOPOULOS, P. & JOHNSON, P., Eds., Design Specification and Verification of Interactive Systems '98, Springer, 102-133.

MARKOPOULOS, P., ROWSON, J. & JOHNSON, P. (1997). Composition and Synthesis with a Formal Interactor Model, *Interacting with Computers*, 9, 197-223.

MARKOPOULOS, P., SHRUBSOLE, P., DE VET (1999). Refinement of the PAC model for the component-based design and specification of television based interfaces. In DUKE, D. & PUERTA, A., Eds., Design Specification and Verification of Interactive Systems '99, Springer, 117-132.

MARKOPOULOS, P., WILSON, S. & JOHNSON, P. (1994). Representation and Use of Task Knowledge in a User Interface Design Environment. *IEE Proceedings~E*, Computers and Digital Techniques, 141, 2, 79-84.

MILNER, R., (1989) Communication and Concurrency. Prentice Hall, UK.

VISSERS C.A., SCOLLO G., VAN SINDEREN M. & BRINKSMA E. (1991). Specification styles in distributed systems design and verification. *Theoretical Computer Science*, 89, 179-206.

WILSON, S. & JOHNSON, P. (1996). Bridging the generation gap: From work tasks to user interface designs. In VANDERDONCKT, J., Ed., *2nd International Workshop on Computer-Aided Design of User Interfaces, CADUI'96*, Presses Universitaires de Namur, 77-94.

13 Appendix : Some elements of LOTOS

This appendix introduces a few elements of the LOTOS language. The reader is referred to (Bolognesi and Brinksma 1989) for a full tutorial. LOTOS is a hybrid specification language that consists of two component languages: a process algebra for the specification of the temporal ordering of the behaviour of systems and the ACT-ONE abstract data typing language. A system is described in LOTOS as a set of interacting processes which interact via *gates*. A simple process-definition is formulated as:

```
process <identifier> [<gate list>](parameter list):<functionality> :=
 <behaviour expression>
where
 <type-definitions>
```

Name	**Syntax**	**Semantics**
inaction	stop	
termination	exit	$exit \xrightarrow{\tau} stop$
action prefix	g;B	$g;B \xrightarrow{g} B$
choice	$B_1[]B_2$	if $B_1 \xrightarrow{\alpha} B_1'$ then $B_1[]B_2 \xrightarrow{\alpha} B_1'$ if $B_2 \xrightarrow{\alpha} B_2'$ then $B_1[]B_2 \xrightarrow{\alpha} B_2'$
disabling	$B_1[>B_2$	if $B_1 \xrightarrow{\alpha} B_1', \alpha \neq \tau$ then $B_1[> B_2 \xrightarrow{\alpha} B_1'[> B_2$ if $B_1 \xrightarrow{\tau} B_1'$ then $B_1[> B_2 \xrightarrow{\tau} B_1'$ if $B_2 \xrightarrow{\alpha} B_2'$ then $B_1[> B_2 \xrightarrow{\alpha} B_2'$
hiding	hide G in B	if $B \xrightarrow{\alpha} B', \alpha \in \mathrm{G}$ then $\mathrm{hide\ G\ in\ B} \xrightarrow{\tau} \mathrm{hide\ G\ in\ B}'$ *if* $B \xrightarrow{\alpha} B', \alpha \notin \mathrm{G}$ then $\mathrm{hide\ G\ in\ B} \xrightarrow{\alpha} \mathrm{hide\ G\ in\ B}'$
renaming	B[H]	if $B \xrightarrow{\alpha} B'$ then $\mathrm{B[H]} \xrightarrow{\mathrm{H}(\alpha)} B'[H]$
parallel composition	$B_1 \mid [G] \mid B_2$	if $B_1 \xrightarrow{\alpha} B_1', \alpha \notin \mathrm{G} \cup \{\tau\}$ then $\mathrm{B}_1 \mid [\mathrm{G}] \mid \mathrm{B}_2 \xrightarrow{\alpha} \mathrm{B}_1' \mid [\mathrm{G}] \mid \mathrm{B}_2$ if $B_2 \xrightarrow{\alpha} B_2', \alpha \notin \mathrm{G} \cup \{\tau\}$ then $\mathrm{B}_1 \mid [\mathrm{G}] \mid \mathrm{B}_2 \xrightarrow{\alpha} B_1 \mid [\mathrm{G}] \mid \mathrm{B}_2'$ if $B_1 \xrightarrow{\alpha} B_1', B_2 \xrightarrow{\alpha} B_2', \alpha \in \mathrm{G} \cup \{\tau\}$ then $\mathrm{B}_1 \mid [\mathrm{G}] \mid \mathrm{B}_2 \xrightarrow{\alpha} \mathrm{B}_1' \mid [\mathrm{G}] \mid \mathrm{B}_2'$

Table 3. Syntax and semantics of the LOTOS operators used in the paper.

```
  <process-definitions>
endproc
```

The gate list specifies the formal identifiers for the gates, i.e. the interaction points through which the process may communicate with its environment. The parameter list refers to state parameters which are specified in ACT-ONE. <functionality> can be *exit* for a terminating process, otherwise it is designated as *noexit*. Behaviour expressions specify the sequencing of interactions. The simplest behaviour expression is a process instantiation (see table 3). More complex behaviour expressions are built up by the composition of process instantiations with LOTOS operators.

A process may specify its participation in an interaction with an action declaration. This can be just a gate identifier if no data communication is specified. It may involve an output of some value, e.g., g!v where v is a value. A LOTOS action declaration may also take the form g?x:t ,where x is a name of a variable and t is a sort identifier indicating the domain of values over which x ranges. This corresponds to a set of possible actions for the behaviour expression. For example, g?x:integer specifies a set of actions g<v> where <v> is in the domain of the integers. Suppose processes A and B are composed in parallel over a gate g as in A | [g] | B. If A offers a value over that gate, e.g., g!true and B offers any event g?x:Boolean, then the value true is passed from A to B.

Table 4 illustrates some notation for describing the transitions and traces of LOTOS processes, where $s \xrightarrow{\mu} r$ denotes a transition from a state s to a state r which is labelled by an action μ.

A relation $R \subseteq Q \times Q$ is a *strong bisimulation relation* on Q iff $\forall (P,S) \in \mathrm{R}, \forall \alpha \in A \cup \{\tau\}$ the following holds:

$$P \xrightarrow{\alpha} P' \Rightarrow \exists S' \mid (P',S') \in R \bullet S \xrightarrow{\alpha} S'$$
$$S \xrightarrow{\alpha} S' \Rightarrow \exists P' \mid (P',S') \in R \bullet P \xrightarrow{\alpha} P'$$

Two processes P and S are called strong bisimulation equivalent if there exists a strong bisimulation relation R relating their initial states, i.e. $(p_0,s_0) \in R$. By substituting $\xrightarrow{\alpha}$ with $\Rightarrow^{\alpha}$ in the definitions above we obtain the *weak bisimulation relation* and the *weak bisimulation equivalence*, denoted as $P \approx Q$.

Notation	Meaning
$q \xrightarrow{\mu_1 .. \mu_n} r$	$\exists q_1 \ldots q_n \in Q \mid q \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} q_2 \rightarrow \ldots \rightarrow q_{n-1} \xrightarrow{\mu_n} r$
$q \Rightarrow^{\varepsilon} r$	$q \equiv r \vee \exists n \geq 1 \mid q \xrightarrow{\tau^n} r$
$q \Rightarrow^{\mu} r$	$\exists q_1, q_2 \in Q \mid q \Rightarrow^{\varepsilon} q_1 \xrightarrow{\mu} q_2 \Rightarrow^{\varepsilon} r$
$q \Rightarrow^{\mu_1 .. \mu_n} r$	$\exists q_1 .. q_{n-1} \in Q \mid q \Rightarrow^{\mu_1} q_1 \Rightarrow^{\mu_2} \ldots \Rightarrow^{\mu_{n-1}} q_{n-1} \Rightarrow^{\mu_n} r$
$q \not\Rightarrow^{\mu_1 .. \mu_n}$	$\not\exists r \in Q \mid q \Rightarrow^{\mu_1 .. \mu_n} r$
Tr(q)	$\{\sigma \in A^* \mid \exists r \in Q \bullet q \Rightarrow^{\sigma} r\}$

Table 4. Some notation to describe traces of processes. Traces are sequences of observable actions which describe transitions from the initial state q of a process to some other state r.