

Composition and synthesis with a formal interactor model

Panos Markopoulos*, Jon Rowson, Peter Johnson

*Department of Computer Science, Queen Mary and Wesfield College, University of London, Mile End Road,
London E1 4NS, UK*

Abstract

This paper discusses the formal specification of interactors, which are primitive abstractions of user interface software, and focuses on the formal aspects of their composition. The composition of interactors is discussed formally in the framework of the Abstraction-Display-Controller (ADC) interactor model. The ADC model has been defined as a LOTOS specification template tailored for specifying user interface software. LOTOS behaviour expressions combining instances of this template specify the composition of interactors to model complex user interfaces. Synthesis is defined as a transformation of these behaviour expressions which supports the generic structure of the ADC model while preserving the meaning of the specified behaviour. Further, the notion of abstract views of interactors is introduced. It is shown how abstract views are themselves primitives for specifying complex interface architectures. © 1997 Elsevier Science B.V.

Keywords: User interface software; Formal specification; Interactors; Composition; Transformations

A central issue concerning research in the formal aspects of human computer interaction is to establish appropriate abstractions of user interface software. Various formal abstractions of interactive systems have been proposed which range from the very abstract, e.g. Harrison and Dix [1] or Dix [2], to more constructive and design oriented representations, e.g. Duke and Harrison [3] and Paternó and Faconti [4]. This paper discusses a formal model of user interface software, which combines lessons learnt from such early formal models of interactive systems with aspects of architectural abstractions of user interface software. This model is called the Abstraction-Display-Controller (ADC) interactor model, and it was introduced in Markopoulos [5] as a template process definition in the LOTOS formal specification language [6].

The general concept of an interactor and its relevance to the implementation of user interface software is discussed in the next section. The ADC interactor model is introduced first informally and then formally as a LOTOS process definition. The elements of the

* Corresponding author.

LOTOS formal specification language used in this paper are summarised in the Appendix A. A range of ADC interactor compositions are supported by LOTOS process algebra operators and their meaning in the context of user interface architectures is discussed. Formal expressions which specify the composition of interactors can be manipulated to formulate larger scale instances of the model. This property is called here the compositionality of the ADC interactor formal model and it is formalised by a transformation called synthesis. Synthesis is defined formally and it is exemplified with an extract from an extensive case study in the use of the ADC interactor, which is reported in Markopoulos [7]. The transformation is shown to preserve the observable behaviour specified. Further the notion of abstract specifications of interactors is introduced. The discussion section at the end of the paper relates this work to current research.

1. Interactors: definition of the concept

Interactors are abstractions which help model interactive systems. Compared to the *earlier abstract models of interactive systems* as, e.g. the state-display model [1] and the PIE model [2], interactors are more structured in that they model interactive systems as compositions of independent entities. Faconti [8] 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 which mediates 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.

Duke and Harrison define an interactor as

...a component in the description of an interactive system that encapsulates a state, the events that manipulate the state and the means by which the state is made perceivable to the user of the system.

The difference between the two definitions lies in the scope of the models, i.e. whether the whole interactive system is modelled or just the user interface. In the latter case it is assumed implicitly that there is at least a *conceptual separation* between the user interface and an underlying application (functional core). The two definitions of the interactor concept and their corresponding formal models are not inconsistent, but they have different origins and they are intended primarily for different purposes. A comparison of the two interactor models, reported in Duke et al. [9], concludes that the interactor model of Paternó and Faconti is better suited as a formal constructive design notation, while the model of Duke and Harrison is better suited for analytical use.

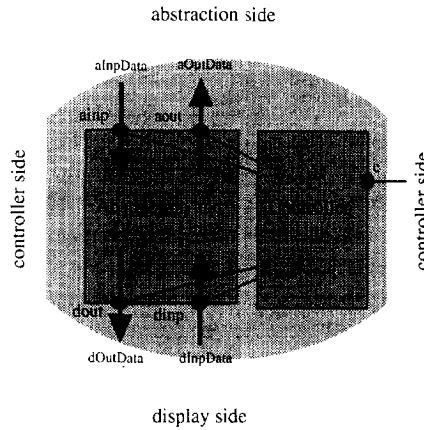


Fig. 1. The internal structure of an interactor.

The term *interactor* has been used also to refer to implementation constructs by Myers [10]. In the context of user interface software architecture the term interactor refers to objects which are characterised by a display function and, in general, support both input and output. While they did not use the term ‘interactor’, the MVC [11] and the ALV [12] architectures support the notion of an interactor as an architectural construct. The PAC architecture [13] supports this notion at a conceptual level. These are object based software architectures, which prescribe the content of the interactors, their role and how they can be composed to build a user interface system. A common characteristic of these models is that interactors are formed as a triplet of smaller scale components. The exact purpose and representation of these components varies across the models, but a coarse description of their purpose is to describe the display of the interactor, its internal (non-displayed) state and its dynamic behaviour. An interesting feature of the PAC and ALV architectures is how the basic interactor structure applies to the user interface as a whole as well as each individual component. This is an appealing concept because it means that such architectural models can help conceptualise a design at varying levels of abstraction and they encourage the modular development of the software. The synthesis transformation which is described in this paper supports this idea in the context of a formal specification.

2. The ADC interactor model

ADC interactors are abstractions of software components which support the communication between the functional core of an interactive system and its user. The user interface system as a whole can be modelled as a single (monolithic) ADC interactor which communicates directly with an application and a user. Alternatively, the ADC model applies to a very small component of the user interface software in which case the ADC interactor adheres to the definition by Faconti quoted in the previous section. In this latter case, the interface can be thought of as a composition of many ADC interactors. In conceptualising a design it might be beneficial to switch between these two views.

An ADC interactor encapsulates two state components—the abstraction and the display. The display models the appearance of an interactor on the screen. The display is output directly to the screen or to some other interactor which will process and output this information. The abstraction holds state information resulting from the interpretation of input already received by the interactor. This in turn, may be interpreted by the interactor to provide input to the application or to other interactors it is connected to. A dedicated component, which is called the Abstraction-Display Unit (ADU), describes how interactions with the environment affect these state components. The temporal ordering of interactions is described separately in the Controller Unit (CU). Fig. 1 illustrates how the ADC interactor is formed by the composition of these two components: the CU controls the occurrence of interactions on all the gates of the ADU.

Interactors can interact with each other and possibly with the user or the application. The interface can therefore be thought of as a graph whose nodes are the interactors and whose edges correspond to connections between them, with the application or the user. ADC interactors are represented in diagrams as barrel shaped nodes, with the top or bottom arch dedicated to connections to the abstraction or display side respectively (Fig. 1). The vertical sides are dedicated to controller connections.

Interactions are distinguished by whether they are associated with input or output of data, pure synchronisation, and by their effect on the state components of the interactor. The concept of a gate has been borrowed from LOTOS to group and to ‘localise’ interactions with a similar purpose. Each gate is associated with one of the sides of the interactor. Gates on the abstraction and display sides correspond to interactions which modify the state parameters of the interactor. When both the display state and the abstraction state are needed to interpret received input, this is considered to arrive at the display side. Intuitively, this corresponds to user input that the interactor needs to de-reference with respect to the current display. In practice this is the only difference between input arriving to one side or the other. Choosing which side an interactor gate belongs to does not depend on what it connects to but on how the information received should be interpreted. Note that the interpretation of input based on the display contents characterises graphical interaction. Output from the display side communicates the value of the display state parameter. Interactions on gates allocated to the controller sides have no direct effect on the state parameters of the interactor.

The ADU which encapsulates the state parameters is ‘neutral’ with respect to the temporal ordering (alias the dialogue) of the interaction, i.e. it is always ready to receive input and to send output on all its gates. The dialogue is modelled in the Controller Unit (CU). The CU is a description of the allowed sequences of actions. It is possible that an interactor has a controller component only. The purpose of such a component will be to coordinate the operations of other interactors and to act as a constraint for the combined behaviour of other interactors.

The description so far has been independent of any particular specification notation. Indeed the model is meant to convey general ideas about the structure of an interface specification. It advocates the separation of dialogue and data handling into distinct components of the interactor and is intended to support a high level description, at least partly represented in a visual form that could be used by interface designers to construct architectural models of an interface. Current work has identified reusable sub-components

of these interactors. This paves the way for tool support for the partly automated generation of formal specifications from more economical visual representations and libraries of specification templates. From the discussion so far it is clear that the formal specification language which would support such a scheme needs to provide both for the specification of dialogue and data handling. This work has been developed using the LOTOS specification language but it is hoped that the concepts above could be implemented using other languages with similar capabilities.

2.1. Formal representation of ADC interactors

The ADC formal interactor model distinguishes three orthogonal aspects of the interactor, which are described in distinct modules:

- The data held by the interactor and the operations upon the data. This is specified by an ACT-ONE abstract data type called the *Abstraction-Display* (AD) data type.
- The link between actions that describe the interactor behaviour and their effect on the data. This is described by a LOTOS process which is called the *Abstraction and Display Unit* (ADU).
- The temporal ordering of the behaviour of the interactor. This is described by a LOTOS process without any local state parameters which is called the *Controller Unit* (CU).

An ADC interactor is formed, in general, by the parallel composition of the ADU and the CU:

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

The ADC interactor is a non-terminating process (therefore the keyword *noexit*). ADU is associated with a set of input–output gates G_{io} . The CU synchronises with the ADU at gates G_{io} but also offers interactions on a gate set G_c . The state parameters of the ADU, listed in round brackets, are initialised with values specified in data type AD. The specification of the data type AD is not discussed further in this paper which focuses on the process algebraic specification of ADC interactors. The reader can find a brief presentation of the data type AD in Ref. [5] and a more extensive discussion in Ref. [14] which discusses also ADC interactors which consist only of a CU or have a different set of state parameters.

2.2. An example of an interactor specification

Fig. 2 shows a typical slider interactor, which provides random access to an order set of data, e.g. a sequence of characters, or static images composing a movie, etc. The thumb indicated on the slider can be modelled as an ADC interactor, which is illustrated in Fig. 2 and is specified as follows:

```
process thumb[press, move, release, doutTHMB, ainpTHMB, aoutTHMB]: noexit: =
  aduTHMB[press, move, release, doutTHMB, ainpTHMB, aoutTHMB](indicator,
  thumb, thumb)
```

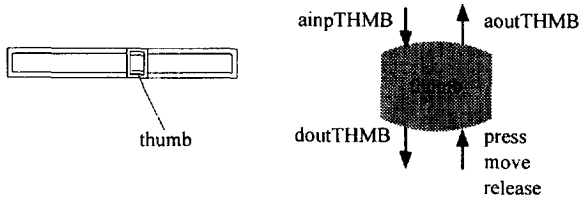


Fig. 2. The thumb interactor and its illustration as an ADC interactor.

```
[[press, move, release, doutTHMB, ainpTHMB, aoutTHMB]]
cuTHMB[press, move, release, doutTHMB, ainpTHMB, aoutTHMB]
endproc
```

The instantiation of the ADU above initialises the state parameters with values (*indicator* and *thumb*) which must be specified in the data type AD. The ADU maps interactions on its gates to the appropriate operations of the data type AD, forging a standardised mapping between the data typing and the process algebraic components of the LOTOS specification. The ADU of the example has three input gates on the display side where *press*, *move* and *release* mouse events are input. With each interaction, the ADU instantiates itself recursively modifying its state parameters by applying operations upon them. These operations which are not detailed here are specified in the corresponding data type AD. In the specification extracts below ‘...’ replaces a gate list where this is obvious from the context.

```
process aduTHMB[...] (a:pnt, dc, ds: thumb_dsp): noexit :=
  aoutTHMB!result(a); aduTHMB[...] (a, dc, ds) []
  doutTHMB!dc; aduTHMB[...] (a, dc, dc) []
  ainpTHMB?x:playBar; aduTHMB[...] (receive(a, x), render (dc, x), ds) []
  press?x:pnt; aduTHMB[...] (inputPress(x, ds, a), echoPress(x, ds, a), ds) []
  move?x:pnt; aduTHMB[...] (inputMove(x, ds, a), echoMove(x, ds, a), ds) []
  release?x:pnt; aduTHMB[...] (inputRelease(x, ds, a), echoRelease(x, ds, a), ds)
endproc
```

The CU for the interactor *thumb* is defined below by the synchronous composition of partial constraints of its behaviour, using the constraint oriented style [15]. The process *inp* describes dragging and *trigger* describes how display interactions follow mouse input, or interactions at gate *ainpTHMB*.

```
process cuTHMB[press, move, release, doutTHMB, ainpTHMB, aoutTHMB]:
  noexit :=
  inp[press, move, release]
  [[press, move, release]]
  trigger[press, move, release, doutTHMB, aoutTHMB, ainpTHMB]
endproc
process inp[press, move, release]: noexit :=
  press?x:pnt; (repeat[move] [>release?x:pnt; inp[press, move, release]])
endproc
```

```

process repeat[move]: noexit: =
move?x:pnt; repeat[move]
endproc
process trigger[press, move, release, doutTHMB, aoutTHMB, ainpTHMB]: noexit: =
(choice X in [press, move, release]
[ ]X?y:pnt; doutTHMB?z:thumb_dsp; aoutTHMB?q:pnt; trigger[...] []
(ainpTHMB?x:playBar; doutTHMB?z:thumb_dsp; trigger[...]
endproc

```

2.3. Topology of interactor gates

The next few paragraphs define the ADC interactor more rigorously. Gate lists rather than gate sets are the standard construct of LOTOS. However the following discussion describes process instantiations using sets of gates. The manipulations and definitions of behaviour expressions that follow attach no significance to the order in which gates are declared in a process instantiation. In practical terms, to support this convention using LOTOS, the recursive process instantiations of the ADU and the CU should preserve the naming and ordering of gates in the process headings. For example, consider the process definition whose heading is:

```
process ADU[dinp, dout, ainp, aout]: noexit: =
```

The convention introduced for the definition of process ADC means that in the body of the process definition it is not allowed to write a recursive instantiation of the forms:

```
ADU[dout, dinp, ainp, aout] or ADU[dinp, dinp, ainp, aout]
```

With the exclusion of such recursive instantiations the gate set $G_{io} = \{dinp, dout, ainp, aout\}$ describes sufficiently the gate-list for the process ADU above.

The set G of the gates of the interactor is partitioned into the set of input–output gates G_{io} and the control gates G_c . All gates of the ADU in Fig. 1 belong to G_{io} and G_c contains a single gate labelled c .

$$G = G_c \cup G_{io} \text{ and } G_c \cap G_{io} = \emptyset$$

Control gates effect actions which have no effect on the state parameters of the interactor and which concern only the controller unit. G_{io} is partitioned to two gate sets G_{abs} and G_{dsp} that correspond to gates on the abstraction and display side of the interactor respectively. For example $G_{abs} = \{ainpTHMB, aoutTHMB\}$ in Fig. 2.

$$G_{io} = G_{abs} \cup G_{dsp} \text{ and } G_{abs} \cap G_{dsp} = \emptyset$$

$$G_{abs} = G_{aout} \cup G_{ainp} \text{ and } G_{aout} \cap G_{ainp} = \emptyset$$

$$G_{dsp} = G_{dinp} \cup G_{dout} \text{ and } G_{dinp} \cap G_{dout} = \emptyset$$

Each of the gate sets G_{dinp} , G_{dout} , G_{ainp} , G_{aout} and G_c corresponds to what can be called a different *role* of a gate for the interactor. The role of a gate determines its use in the ADU and the CU. The definition of ADC in the following paragraphs, specifies actions on a

particular gate according to which of the gate sets above it belongs to. For convenience, the role of a gate will be referred to simply as *dinp*, *dout*, *aout*, *ainp* and *c*. Apart from the gates, the set of actions offered by an interactor depends on the data offered on its G_{io} gates. It is not always necessary that an ADC interactor specification should be associated with data type AD even if it models data input or output. What is important for this discussion is that interactions on output gates actually offer a value and that interactions on input gates do not refuse a value offered to them. Value offers may be either the display state of the interactor or an interpretation of the abstraction by an inquiry operator. Provided that the AD is internally consistent there will always be some value output on the output gates and by definition a non empty set of interactions will be offered on input gates.

2.4. Elementary ADU

An elementary ADU is a recursive non-terminating process for which the following holds:

1. Its gate set G_{io} can be partitioned in a set of input gates G_i and a set of output gates G_o , such that $G_{io} = G_i \cup G_o$ and $G_i \cap G_o = \emptyset$.
2. Interactions on input gates that involve the communication of data are strictly variable declarations without a selection predicate (see definitions in Appendix A).
3. Output actions are strictly value declarations. They are either the value of a local parameter, e.g. the display state, or the value of an enquiry operator on the abstraction state parameter of the ADU. In order that a value is always offered the data type AD which corresponds to the elementary ADU should be internally consistent.
4. The elementary ADU is a behaviour expression that offers a choice of events on all the gates of the ADU before instantiating itself recursively. The recursive instantiation updates the local variables by applying the operations of the data type AD which corresponds to the role of each gate.

2.5. A well formed ADU

A well formed ADU may be:

1. An elementary ADU.
2. A parallel composition expression of the form $ADU_A[G_{io}^A][G]ADU_B[G_{io}^B]$ where ADU_A and ADU_B are well formed ADUs and where $G_o^A \cap G_o^B \cap G = \emptyset$.
3. The following is stipulated for the gates of the well formed ADU: $G_{io} = G_i \cup G_o$, where $G_i = G_i^A \cup G_i^B$ and $G_o = (G_i^A \cup G_i^B) - G_o$.

The parallel composition of two ADUs that synchronise on some of their gates is also a well formed ADU. Requirement 2 ensures that it is not possible to specify an ADU that might deadlock because of its two components offering different values on the same output gate. Requirement 3 describes the definition of the gate sets for the composite ADU. A gate which is an input gate for one of the components and an output gate for the other is classified as an output gate for the composite expression. This ensures that

the input and output gates of the derived ADU do not intersect, i.e. $G_i \cap G_o = \emptyset$, in accordance to the topology of the interactor gates. Further, it is clear that $G_{io} = G_{io}^A \cup G_{io}^B$.

2.5.1. Lemma. Behaviour of a well formed ADU

The behaviour of a well formed ADU is characterised by the following:

1. The set of interactions offered at each gate is not empty.
2. $P(\text{ADU}) \sim Q$, with $Q = \text{choice } g \text{ in } [G_{io}][]g; Q$,

where \sim denotes strong bisimulation equivalence (see Appendix A) and $P(\cdot)$ denotes the naive transformation from full LOTOS to basic LOTOS, which maps each specified interaction $g \langle v \rangle$ to some interaction g on the same gate but which ignores the data component of the action specification (see Appendix A).

Process Q is a basic LOTOS process which is always ready to offer interactions on all its gates. Property 2 states that if the data values associated with interactions on the gates of the ADU are ignored, then its behaviour may be described by Q . This property is a rigorous expression of the fact that the ADU does not model the temporal behaviour of the interactor. This lemma follows by structural induction and by the distributivity of the naive transformation $P(\cdot)$.

2.6. The controller unit (CU)

The temporal ordering of the interactions of the ADU is described in the CU. This is a recursive process with functionality *no-exit*. For the purposes of this discussion it can be any LOTOS behaviour expression with gates $G_{io} \cup G_c$. The CU is a process such that:

1. Gates in G_{io} are variable declarations whose sorts are determined by the typing of the gates in the ADU.
2. CU does not output any values, i.e. it does not specify interactions with a value specification component.
3. CU does not have any selection predicates on actions specified with a variable declaration.
4. CU does not have any local state variables.

Conditions 1–3 ensure that the CU is consistent with ADU. Condition 4 ensures that the CU pertains strictly to the temporal ordering of interactions and not to the communication and manipulation of data. The definition does not constrain the required behaviour of the CU which encodes the dialogue supported by the interactor. Properties 1–4 restrict the use of LOTOS constructs, affecting the ease with which a dialogue may be specified.

2.7. Formal definition of ADC interactors

A LOTOS behaviour expression is an ADC interactor specification if:

1. it is a well formed CU; or

2. it is the synchronous composition of a well formed ADU and a well formed CU, with the two component processes synchronising on all the gates of the ADU.

3. Composition of interactor specifications

LOTOS operators can help specify complex behaviours by composing ADC interactors. This section examines the meanings that such compositions might have. Consider two interactors ADC_A and ADC_B with respective gate sets G_A and G_B . The operator $[[G]]$ describes the partial synchronisation of ADC_A and ADC_B over a gate set $G \subseteq G_A \cap G_B$. Full synchronisation $||$ and pure interleaving $|||$, can be thought of as boundary cases of the partial synchronisation of behaviour expressions, where $G = G_A \cup G_B$ and $G = \emptyset$ respectively. The notion of the role of a gate was introduced in paragraph 2.3. This may be the role *dinp*, *dout*, *ainp*, *aout*, or *c*. A gate $g \in G$ may have different roles for each of the two interactors. A different *type of connection* is obtained for each combination of roles, e.g. (*dinp*, *dout*). This discussion does not distinguish between interactors ADC_A and ADC_B , so a connection type is symmetric, i.e. the same type of connection corresponds to the combinations (*dinp*, *dout*) and (*dout*, *dinp*). The parallel composition operator $[[G]]$ may specify no connection (pure interleaving) or several of the connection types (a)–(l) below (see also Fig. 3):

- (a) Connection type (*dinp*, *dinp*). Both interactors receive data synchronously from their display side. An example is a multiple selection of icons which are ‘dragged’. In this case the mouse position is read by both interactors.
- (b) Connection type (*ainp*, *dinp*). As with type (a) the two interactors are synchronised consumers of data. In this case though, the input arrives at the abstraction side for interactor A which interprets the input data independently of the display.
- (c) Connection type (*dout*, *dinp*). This type of connection models the reuse of the graphical output from interactor A as graphical input to interactor B. For example, an interactor may ‘capture’ the instantaneous graphical output of a graphical application.
- (d) Connection type (*aout*, *dinp*). Data is sent from the abstraction side of A to the display side of B. For example, the *thumb* interactor of Fig. 2 should send its result to an interactor modelling the whole slider.
- (e) Connection type (*c*, *dinp*). The controller unit of interactor A and the input on the display side of interactor B are mutually constrained. This could be used to implement a mode, e.g. a keyboard modifier. For example, interactor A can model the keyboard. Interactor B will receive input at gate *dinp* only when interactor A synchronises with it, which will be only when the appropriate key is pressed.
- (f) Connection type (*ainp*, *ainp*). In this case also, the two interactors are synchronous consumers. Consider a graphical interface, which is resized following a menu command. One possible approach to modelling the interface is that interactors should receive the new screen coordinates of their enclosing window from their abstraction sides.
- (g) Connection type (*dout*, *ainp*). Data is sent from the display side of B to the abstraction side of A. This could be an example of a graphics output pipeline, where each interactor manages one transformation of the graphics data structures.

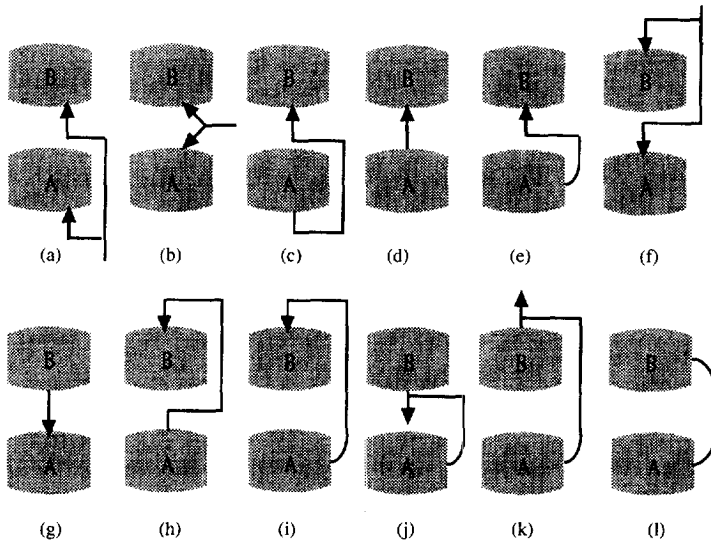


Fig. 3. The range of connection types of ADC interactors. Arrows indicate gates for input or output. The last case represents a simple synchronisation.

(h) Connection type (*aout*, *ainp*). A value is communicated from A to B. For example, the slider of Fig. 2 may compute a numeric value which it then sends to other interactors as input on their abstraction side. For example, a slider could help select from an ordered sequence of data elements by using this number as the offset of the selected element.

(i–l) Connection types (*c*, *ainp*), (*c*, *dout*), (*c*, *aout*), and (*c*, *c*). In these cases the controller of interactor A constrains, and is constrained by, a gate of interactor B. This is similar to type (e) above, where interactions on the common gate must satisfy the conjunction of constraints specified by the CUs of the two interactors.

The list above does not include all the possible connection types. Fig. 4 illustrates those omitted. These connection types concern pairs of interactors which synchronise over common output gates and thus may deadlock when the two interactors attempt to output a different value on their common gate. Excluding these cases means that a deadlock can be effected only by inconsistent dialogues and not by inconsistent data values. This ensures that the dynamic behaviour is not specified implicitly by the ADUs.

Choice can be used to specify alternative interactions. Consider, for example, a set of interactors for drawing different shapes on a drawing package which are invoked by some interactor supporting logical disjunction, e.g. a palette or a set of radio buttons. The alternative interactors can be related by the choice operator and their composition could be synchronised with the menu interactor on their initial events. Disable can be used to 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 an 'ok' or a 'cancel' button and by composing the two with the disable operator of LOTOS. Choice and disable are easily recognised as useful constructs for specifying human-computer dialogues. They

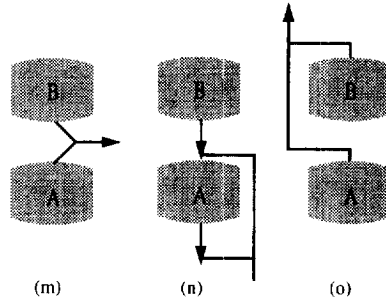


Fig. 4. The types of connections which have been ruled out.

are called dynamic operators because the transitions they specify result in behaviour expressions with a different syntactic structure than that of the original expression. This makes them particularly suitable for specifying the dynamic deactivation of interactors and in conjunction with the parallel operators for describing the dynamic activation of interactors. Mezzanotte and Paternó [16] examine the effect these operators have on the expressive power of an interactor model and consequently in the feasibility of verifying interface specifications. The combination of dynamic composition operators with parallel composition expressions can result in non-finite interpretations of the specification which makes the verification by model checking infeasible. This trade-off between expressive power and the uses of the model needs to be investigated with further practical applications of the model.

4. Synthesis of ADC interactors

4.1. Formal definition of synthesis

Complex interfaces can be specified as a LOTOS expression using the compositions discussed above. What we mean by compositionality is that it should be possible to re-write such a LOTOS expression into a single compound ADC interactor, i.e. to a single LOTOS process which satisfies the conditions described earlier. This rewriting transformation is termed *synthesis* of interactors (Fig. 5). The synthesis transformation is defined by:

1. An input expression, which is called the distributed form: $DF = ADC_A \otimes ADC_B$ where \otimes is a binary composition operator for non-terminating LOTOS behaviour expressions (one of $||$, $|||$, $[>$, $[]$ and $||G||$ where $G \subseteq G_A \cap G_B$).
2. An output expression which is called the compound form: $CF = ADU_{AB} || [G_{io}^A \cup G_{io}^B] || CU_{AB}$ where $ADU_{AB} = ADU_A || [G_I] || ADU_B$ and $CU_{AB} = CU_A \otimes CU_B$. This form is determined by \otimes and G_I .
3. The *transformation requirement* is that the compound form is itself an ADC interactor. The expression CF is an ADC interactor, provided the ADU_{AB} is a well formed ADU. It is therefore necessary that: $G_I \cap G_0^A \cap G_0^B = \emptyset$.

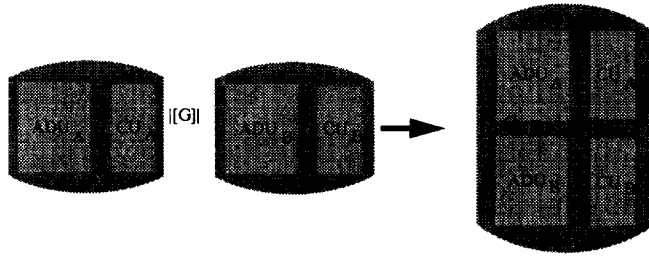


Fig. 5. Synthesis applied to the synchronous composition of two interactors.

4. The *correctness preservation requirement* describes the invariant of the transformation, i.e. it expresses the desired relationship between the semantics of the distributed form and the compound form. Clearly some congruence relation is required, i.e. the two forms should not be distinguished in the various algebraic contexts they might appear in. Strong bisimulation equivalence is proven below which entails the weaker requirement of congruence (see Appendix A).

4.2. Example

The ADC model has been tested by developing a specification of the graphical interface of Simple Player™, an application program for the Macintosh computer which gives access to play-back and editing facilities for QuickTime™ movies. The specification was elaborated to an abstraction level much lower than would be normally intended in a formal specification of an interface design. For the purposes of the case study, this approach provides added confidence in the validity of the model: the comparison with the observed behaviour is straight-forward and the contingencies of the represented interactions are not brushed over or abstracted away from. A full exposition of the case study and its results may be found in Ref. [7]. Here we present just a small part of it. Each of the visible components of the Simple Player™ interface was modelled as an ADC interactor. The interface was modelled as a composition graph of these interactors, plus some auxiliary interactors used as connectives. A small segment of this graph is shown in Fig. 6. This includes the three interactors indicated on the snapshot of Simple Player™ in operation. The elliptic nodes are special purpose interactors used as ‘connectives’ to communicate data asynchronously between interactors.

The abstraction side of the interactor *playerBar* is connected to the functional core, not shown in Fig. 6. This core sends to the player bar the time coordinate of the currently shown frame of the movie played. The player bar interactor outputs its display state through gate *doutPB*. There are two recipients for this data. The thumb interactor receives the time information at gate *ainpTHMB* through the connective *conn1* shown as an ellipse, and the selection band receives the time information at gate *ainpSB*. The thumb interactor uses this information to update its display, offered at gate *doutTHMB*. Similar information can flow in the opposite direction. Mouse events are input to the thumb from the display side and communicated to the other two interactors from gate *aoutTHMB*, once more with

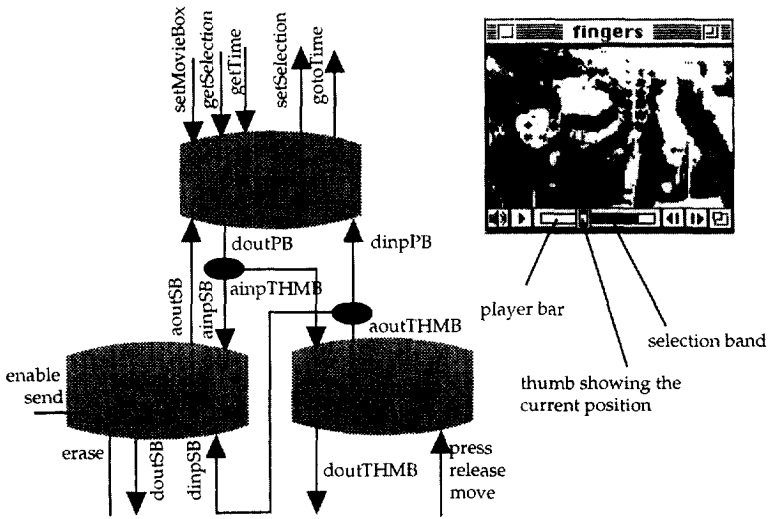


Fig. 6. The composition of three interactors of Simple PlayerTM.

the help of a connective (*conn2*). It is not suggested that the structure shown in Fig. 6 is the structure of the actual software which is defined as a layered set of function libraries. Rather, interactors are proposed as useful abstractions for structuring a formal specification and the particular composition above was found helpful for this purpose in the case study.

It is clear, that the three interactors and the two connectives are very closely related, in fact, they can be thought of as a single interactor. This would model the behaviour of a movie slider that combines the functions of its components, i.e. it allows selections to be made and movie positions to be input or output. In the context of the graph representing the global interface architecture, the graph segment of Fig. 6 could be replaced by a single barrel node. In this process of synthesis a group of interactors is substituted by a single interactor that models the same observable behaviour. The behaviour expression which corresponds to the structure of Fig. 6 (the distributed form) is presented below followed by its rewriting as a single composite interactor (the compound form).

The LOTOS behaviour expression defining the distributed form (DF) is:

```
((playerBar[dinpPB, aoutSB, doutPB, setMovieBox, getSelection, getTime,
gotoTime, setSelection]
|[doutPB]]
conn1[doutPB, ainpTHMB, ainpSB])
|[ainpTHMB, dinpPB]]
(thumb[press, move, release, doutTHMB, ainpTHMB, aoutTHMB]
|[aoutTHMB]]
conn2[aoutTHMB, dinpPB, dinpSB]))
|[dinpSB, aoutSB, ainpSB]]
selectionBand[dinpSB, doutSB, ainpSB, aoutSB, erase, enable, send]
```

The specification of the thumb interactor has been described already. For brevity the specifications of the remaining interactors are not detailed here. The reader is referred to Ref. [7] for a fuller exposition.

Process *compound*, the compound form (CF), has the general ADC structure:

```
process compound[Gio]:noexit :=
  adu[Gio] |[Gio] cu[Gio]
endproc
```

where the following shorthand is used:

```
Gio = {setMovieBox, getSelection, setSelection, getTime, gotoTime, doutPB,
  dinpPB, aoutSB, ainpSB, ainpTHMB, aoutTHMB, press, move, release, doutTHMB,
  dinpSB, doutSB, erase}
```

The ADU and the CU of the compound form are defined as follows (state parameters are instantiated with initial values which are specified in the corresponding data types):

```
process adu[Gio]:noexit :=
  ((aduPB[dinpPB, aoutSB, doutPB, setMovieBox, getSelection, getTime, gotoTime,
  setSelection] (initPlayBarData, thePlayerBar, thePlayerBar)
  |[doutPB]|
  aduCONN1[doutPB, ainpTHMB, ainpSB](thePlayerBar))
  |[ainpTHMB, dinpPB]|
  (aduTHMB[press, move, release, doutTHMB, ainpTHMB, aoutTHMB](indicator,
  thumb, thumb)
  |[aoutTHMB]|
  aduCONN2[aoutTHMB, dinpPB, dinpSB](aPoint)))
  |[dinpSB, aoutSB, ainpSB]|
  aduSB[dinpSB, doutSB, ainpSB, aoutSB, erase](noSelection, thePlayerBar, the-
  PlayerBar)
endproc

process cu[Gio]:noexit :=
  ((cuPB[dinpPB, aoutSB, doutPB, setMovieBox, getSelection, getTime, gotoTime,
  setSelection]
  |[doutPB]|
  cuCONN1[doutPB, ainpTHMB, ainpSB])
  |[ainpTHMB, dinpPB]|
  (cuTHMB[press, move, release, doutTHMB, ainpTHMB, aoutTHMB]
  |[aoutTHMB]|
  cuCONN2[aoutTHMB, dinpPB, dinpSB]))
  |[dinpSB, aoutSB, ainpSB]|
  cuSB[dinpSB, doutSB, ainpSB, aoutSB, erase, enable, send]
endproc
```

This conceptually simple idea, of grouping and reshuffling interactor components, has several implications for the formal model. It raises the question of what it means to group interactors and further what it means for the two specified behaviours to be the same. As

far as the formal specifications are concerned, it is desirable that the two behaviour expressions can be interchanged as components of larger scale specifications without changing the meaning of the overall specification. A definition of whether two specified interfaces are the same from the user's point of view is a harder problem. It depends on how the users perceive and compare user interface behaviours, and whether they are able to discern the kind of differences that a formal comparison of the specifications reveals. Such a theory of user cognition is outside the scope of this paper.

The term 'grouping' is used here to describe the surface structure of a behaviour expression. This structure affects the use of the specification either constructively, as a building block for higher level components, or analytically, e.g. for the verification of the specified dialogue. The synthesis of interactors by such groupings enables the bottom-up construction of interactors from library components, or conversely the top down refinement of abstract specifications to structure closer to the implementation architecture. Supporting this concept of composition, by appropriate abstractions, notations and tools, can get us a step closer to supporting the use of formal methods in the interface design process. Analytically, synthesis can be seen as 'factoring out' CU specifications, which describe the dialogue of the user interface. Regardless of whether there is a separable dialogue component, synthesis enables the separation of a dialogue specification, which can then be analysed separately. Restricting the size of the specification and eliminating the data component of LOTOS facilitates the practical verification of dialogue related properties by model checking tools.

From an interface design perspective synthesis transforms an implementation view of the design as a composition of individual interactors to a design oriented view which separates a centralised dialogue description from data handling for the global interface specification. The latter is easier to inspect and easier to manipulate in order to design interaction and induces a rational separation of concerns for the designer. In terms of LOTOS specifications it amounts to a transition from a resource oriented specification to a constraint oriented specification [15]. In contrast, the conventional specification practice involves the transformation from a constraint oriented view to a resource oriented view. Supporting this transition between specification styles reflects a view that the application of formal methods to the design, specification and verification of user interfaces, should align with the user interface design activity rather than dictate it. Guindon [17] argues that user interface designers work both bottom-up and top-down. In either case, it will always be necessary to construct complex entities by the composition of smaller scale components, regardless of whether the design knowledge flows top-down or bottom-up. In Markopoulos [14] the reverse transformation to synthesis, called the decomposition of ADC interactors, is also introduced as a necessary complement to synthesis to support the stepwise refinement of interface specifications.

5. Correctness of the synthesis transformation

5.1. Correctness for the synchronous composition of interactors

The distributed and the compound forms of the composition are:

$$DF = (ADU_A | [G_{i_0}^A] | CU_A) | [G] | (ADU_B | [G_{i_0}^B] | CU_B)$$

$$CF = (ADU_A \parallel [G_1] \parallel ADU_B) \parallel [G_{i0}^A \cup G_{i0}^B] \parallel (CU_A \parallel [G_2] \parallel CU_B)$$

Strong bisimulation equivalence of DF and CF is proved below, subject to constraints for the gate sets G_1 and G_2 . The proof technique used here is adapted from Vissers [15] and is a ‘shorthand’ version of the proof by bisimulation which is based on the following theorem by Milner [18]:

$DF \sim CF$ if $(DF, CF) \in R$ and R is a strong bisimulation relation.

Because an ADC interactor specifies no ‘silent’ actions i , the following must hold for R to be a bisimulation relation (see Appendix A).

For all interactions $g \langle v \rangle$ for which there is a transition for DF or CF:

$$DF \xrightarrow{g \langle v \rangle} DF' \Rightarrow \exists CF' \mid (DF', CF') \in R \cdot CF \xrightarrow{g \langle v \rangle} CF'$$

$$CF \xrightarrow{g \langle v \rangle} CF' \Rightarrow \exists DF' \mid (DF', CF') \in R \cdot DF \xrightarrow{g \langle v \rangle} DF' \quad (\text{CPR})$$

5.1.1. Proof

A relation R is proposed that can play the role of the bisimulation relation. For (CPR) to hold, necessary and sufficient conditions are derived relating the gate sets of the component processes of DF and CF.

Let

$$R = \{ \langle (A \parallel [S_1] \parallel B) \parallel [S_2] \parallel (C \parallel [S_3] \parallel D), (A \parallel [S_4] \parallel C) \parallel [S_5] \parallel (B \parallel [S_6] \parallel D) \rangle \}$$

where A , B , C , and D are behaviour expressions, S_1 to S_6 are gate sets. Clearly, $(DF, CF) \in R$ by substitution. From the semantics of the synchronisation operator, any transition of DF or CF involves the transition of at least one of its components. Further the transition will always result in the same static structure for DF' and CF' . Thus, $(DF', CF') \in R$, for any transition $g \langle v \rangle$.

Condition (CPR) requires that DF and CF have the same sets of transitions. DF and CF are behaviour expressions formed by the same components, so all possible transitions can be categorised by considering each possible combination of transitions for their components. Transitions are labelled by $g \langle v \rangle$ which is defined by gate identifier g and value $\langle v \rangle$. By the semantics of synchronous composition, the gate identifier g must belong to the gate set of the component process performing the transition which does not change with the recursive instantiation of the processes ADU and CU. The domain of g for each type of transition is the intersection of the gate sets of the component processes, constrained by the behaviour expression DF or CF so that the transition is possible, according to the operational semantics of LOTOS. An empty gate set means that the transition is impossible. To ensure that DF and CF offer the same events, the domains for g of DF and CF are equated. If $L(P)$ stands for the gate set of a process instantiation P , then (CPR) requires that $L(DF) = L(CF)$. Simple set manipulations result in the necessary and sufficient conditions between the gate sets of the two expressions. The results are listed in Table 1. Note that some combinations are impossible by the definition of the model, as for example an ADU firing without synchronising with the CU component that controls it.

For example, consider the first row of Table 1. It describes the case of a transition of the form $ADU_A \rightarrow ADU'_A$. For the DF such a transition is impossible because by the definition of ADC all gates of the ADU synchronise with the gates of its corresponding CU. The second row describes a transition of the form $CU_A \rightarrow CU'_A$. For DF the transition may happen with an interaction at any of the gates in $L(CU_A)$, provided it does not belong to either the gates of ADU_A or the synchronisation gates in G . For the CF the transition may happen only if the gate g is not in the synchronisation gates with CU_B , i.e. the set G_2 , or in the set of gates $G_{i_0}^A \cup G_{i_0}^B$ at which the two controllers synchronise with the two ADUs. By equating the two gate sets the condition listed on the table arises:

$$\begin{aligned} L(DF) = L(CF) &\Rightarrow G_c^A - G_{i_0}^A - G = G_c^A - G_2 - (G_{i_0}^A \cup G_{i_0}^B) \Rightarrow G_c^A \cap G \\ &= G_c^A \cap (G_2 \cup G_{i_0}^B) \end{aligned}$$

In this case a useful condition results. In others a tautology arises which is noted as 'no condition' on the corresponding row of the table. The combination of the resulting conditions allows many possible constructions for G_1 and G_2 of the compound form. The condition that results from their combination is

$$G_{i_0}^A \cap G_{i_0}^B \subseteq G, G_1, G_2 \text{ and } G_1 \cap ((G_{i_0}^A \cap G_c^B) \cup (G_{i_0}^B \cap G_c^A)) = \emptyset$$

To satisfy this condition G_1 and G_2 can be constructed as follows:

$$G_1 = G_{i_0}^A \cap G_{i_0}^B \cap G \wedge G_2 = G$$

This means that the controllers in the CF must be composed over exactly the synchronisation gate set used in the DF, while the ADUs can not synchronise over the gates implementing connections of types (e), (i), (j), (k) and (l) of Fig. 3. By the transformation requirement $G_1 \cap G_0^A \cap G_0^B = \emptyset$ (see paragraph 6.1) and by the construction above it follows that the input form must be such that $G \cap G_0^A \cap G_0^B = \emptyset$. This, as was expected, rules out input forms implementing the connections of types (m), (n) and (o) of Fig. 4.

The discussion so far has ignored the data values passed. It is argued that this does not discredit the proof presented. An interaction may involve the communication of data from one interactor to another, unless it involves both interactors reading data from a third source or simply synchronising. These two cases are examined separately.

Connections (c), (d), (g) and (h) are the only connection types that support data communication. Without loss of generality it is assumed that A produces data values and B consumes them. By the definition of the model, the gates of the ADU are typed. Let $\text{ValueSet}(t)$ denote the possibly infinite set of values of sort t , and $\text{range}_A(g) \subseteq \text{ValueSet}(t)$ (respectively $\text{range}_B(g) \subseteq \text{ValueSet}(t)$) denote the set of values possibly offered by ADU_A (respectively read by ADU_B) over gate g . Predicate selection was ruled out in the definition of the interactors so: $\text{range}_B(g) = \text{ValueSet}(t)$. Also, by definition, CU_A and CU_B allow all values $v \in \text{ValueSet}(t)$. In Table 1, data communication takes place only in the transition in the last row. By inspection of the DF and the CF it is easy to see that in both cases the set of possible values is $\text{range}_A(g)$. Thus, for any gate label g , DF and CF offer the same transitions $g \langle v \rangle$, with $\langle v \rangle \in \text{range}_A(g)$.

Table 1

Each row shows a combination of transitions and the condition put on the label sets of the components and G, G_1, G_2 for it to be possible for both DF and CF

ADU _A	CU _A	ADU _B	CU _B	Condition
•				No resulting condition
	•			$G_c^A \cap G = G_c^A \cap (G_2 \cup G_{io}^B)$
•	•			$G_{io}^A \cap G = G_{io}^A \cap (G_1 \cup G_2)$
		•		No resulting condition
•		•		No resulting condition
	•	•		$(G_{io}^B - G_1) \cap ((G_c^A \cup G_{io}^A) - G_2) = \emptyset$
•	•	•		No resulting condition
			•	$G_c^B \cap G = G_c^B \cap (G_2 \cup G_{io}^A)$
•			•	$(G_{io}^A - G_1) \cap ((G_{io}^B \cup G_c^B) - G_2) = \emptyset$
	•		•	$G_c^A \cap G_c^B \cap G = G_c^A \cap G_c^B \cap G_2$
•	•		•	$G_{io}^A \cap G_c^B \cap G = (G_2 - G_1) \cap ((G_{io}^A \cap G_c^B) \cup (G_{io}^A \cap G_{io}^B))$
		•	•	$G_{io}^B \cap G = G_{io}^B \cap (G_1 \cup G_2)$
•		•	•	No resulting condition
	•	•	•	$G_{io}^B \cap G_c^A \cap G = (G_2 - G_1) \cap ((G_{io}^B \cap G_c^A) \cup (G_{io}^A \cap G_{io}^B))$
•	•	•	•	$G_{io}^A \cap G_{io}^B \cap G = G_{io}^A \cap G_{io}^B \cap G_1 \cap G_2$

The remaining types of connections, i.e. (a), (b), (e), (f), (i), (j), (k) and (l) specify the synchronisation of the two interactors with a variable declaration on both sides of a connection. In this case, both the DF and the CF offer the same set of interactions $g \langle v \rangle$ where g is the synchronisation gate and $\langle v \rangle$ is any value in ValueSet(t).

5.2. Correctness of synthesis for choice and disable

The distributed and the compound forms of the transformation for choice are as follows:

$$DF = (ADU_A || [G_{io}^A] | CU_A) || (ADU_B || [G_{io}^B] | CU_B)$$

$$CF = (ADU_A || | ADU_B) || [G_{io}^A \cup G_{io}^B] | (CU_A || CU_B)$$

The proof of DF~CF follows the same proof technique as for the synchronous composition, but it is simpler as it does not need to consider the communication of data between the interactors. The bisimulation relation is defined to contain two pairs of behaviour expressions with the give forms:

$$R = \{a, b\} \text{ where}$$

$$a = \langle A || [S_1] | B, (C || [S_2] | D), (A || | C) || [S_3] | (B || | D) \rangle \text{ and}$$

$$b = \langle A || [S_1] | B, (A || | C) || [S_2] | B \rangle \}$$

where A–D are behaviour expressions and S_1 – S_3 are gate sets. (DF, CF) is isomorphic to type (a) of elements of relation R, so $(DF, CF) \in R$. To satisfy the definition for the bisimulation relation DF and CF need to have exactly the same sets of transitions and the result of the transition will also be a pair of behaviour expressions belonging to R. The structure of the behaviour expressions changes with the first transition into a pair (DF', CF') of type (b). Transitions for pairs of behaviour expressions (DF, CF) of type (b)

maintain the same structure, so R indeed contains all the possible pairs of behaviour expressions that might result from a transition.

By a process similar to that described in the previous paragraph the following necessary and sufficient conditions are derived so that $DF \sim CF$:

$$G_c^A \cap G_{i_0}^B = \emptyset \wedge G_c^B \cap G_{i_0}^A = \emptyset \wedge G_{i_0}^A \cap G_{i_0}^B = \emptyset$$

By a similar process the corresponding expressions to DF and CF obtained by substituting \parallel with $>$ can be shown to be equivalent under the same conditions (cf. Ref. [14]).

6. Composition and hiding

A LOTOS hide expression designates that parts of some specified behaviour can not be observed by its environment. The hide operator takes as an argument a set of gates which are hidden. Hiding may apply to an interactor or to a group of interactors in order to abstract from internal detail. The term abstract view is adopted here to describe an interactor some of whose gates are hidden:

hide G_h in ADC

Abstract views can be used as building blocks for interface specifications. As was the case with interactor specifications, it is interesting to examine in what circumstances a composition expression whose arguments are abstract views can itself be reshaped into an abstract view. Fig. 7 illustrates the synthesis of an abstract view from the synchronous composition of two abstract views. Abstract views are illustrated as black frames enclosing the interactors-barrels.

Consider a behaviour expression DF which specifies the synchronisation of the abstract views of two interactors ADC_A and ADC_B over a gate set G :

$$DF = (\text{hide } G_h^A \text{ in } (ADU_A \parallel [G_{i_0}^A] \parallel CU_A)) \parallel [G] \parallel (\text{hide } G_h^B \text{ in } (ADU_B \parallel [G_{i_0}^B] \parallel CU_B))$$

where $G \cap G_0^A \cap G_0^B = \emptyset$

Provided that any one abstract view does not hide gates that belong to the gate set of the other, i.e. $G_h^A \cap (G_{i_0}^B \cup G_c^B) = \emptyset$ and $G_h^B \cap (G_{i_0}^A \cup G_c^A) = \emptyset$ synthesis results in the following compound form, which is weak observation congruent to DF:

$$CF = \text{hide } (G_h^A \cup G_h^B) \text{ in } (ADU_A \parallel [G_1] \parallel ADU_B) \parallel [G_{i_0}^A \cup G_{i_0}^B] \parallel (CU_A \parallel [G] \parallel CU_B)$$

where $G_1 = G_{i_0}^A \cap G_{i_0}^B \cap G$.

A similar congruence describes the synthesis of dynamic composition expressions involving abstract views:

$$(\text{hide } G_h^A \text{ in } (ADU_A \parallel [G_{i_0}^A] \parallel CU_A)) \otimes (\text{hide } G_h^B \text{ in } (ADU_B \parallel [G_{i_0}^B] \parallel CU_B))$$

$$\cong \text{hide } G_h^B \cup G_h^A \text{ in } ((ADU_A \parallel \parallel ADU_B) \parallel [G_{i_0}^A \cup G_{i_0}^B] \parallel (CU_A \otimes CU_B))$$

where \otimes is $[] \vee [>$ and $G_c^A \cap G_{i_0}^B = \emptyset \wedge G_c^B \cap G_{i_0}^A = \emptyset \wedge G_{i_0}^A \cap G_{i_0}^B = \emptyset$.

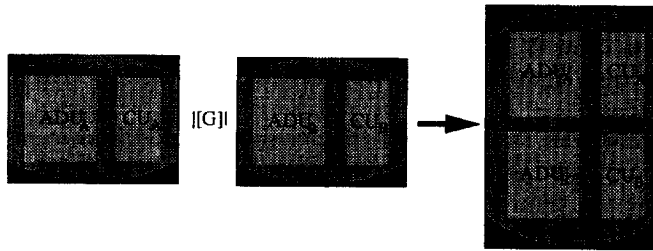


Fig. 7. Synchronous composition of abstract views.

These congruences follow easily from the corresponding equivalences defined for ADC interactors. They show that the notion of an abstract view is an essential extension to that of an interactor. Abstract views support the notion of compositionality provided that the hidden gates of one abstract view and the gate set of the other processes do not overlap. This is not a severe constraint in writing specifications but it does mean that some caution should be exercised in naming the gates of an interactor.

A congruence law for hiding (cf. Ref. [6], p. 90) states that successive applications of hide can be ‘factored out’ of a behaviour expression:

$$\text{hide } A \text{ in } (\text{hide } A' \text{ in } B) \cong \text{hide } A \cup A' \text{ in } B$$

It follows that the cumulative effect of applying synthesis successively to a behaviour expression which involves abstract views and interactors is to rewrite the expression as the abstract view of a compound ADC interactor:

$$\text{hide } [\dots \text{all hidden gates} \dots] (\text{ADU}_{\text{top level}} \parallel \text{G}_{\text{top level}} \parallel \text{CU}_{\text{top level}})$$

where the $\text{ADU}_{\text{top level}}$ is a well formed ADU and the $\text{CU}_{\text{top level}}$ is isomorphic to the original behaviour expression.

7. Discussion

An important motivation in developing the ADC interactor has been to support the systematic re-use of formal specifications components and a principled approach to their composition. The idea of composing interactors has been further developed here. This paper has focused on the use of two LOTOS operators: synchronisation and hiding. Synchronisation specifies the static compositions of interactors, which intuitively corresponds to interactors appearing together on the screen as one object. It can model data communication between interactors and mutual constraints on each other’s dialogue. Hiding helps abstract away from internal detail of compound interactors.

The definition of a formal framework for the modular development of software by the composition of lower level entities or by the refinement and specialisation of higher level entities is a subject of research in object oriented formal specification techniques [19]. As mentioned already, the concept of an interactor is similar to that of an object, considered as

an encapsulated module of specification which interacts with its environment through a prescribed interface and a standard model of communication. In this narrow sense, the ADC interactor is an object based formalism. However, objects are generally created and destroyed dynamically at run-time, and this notion of the lifetime, or existence, of an object with its own identity, should be reflected in an object-based specification. On the other hand, an ADC formal interactor describes the lifetime of an interactor for which the connections and the context for its execution are specified statically and it does not generalise readily to an indefinite number of objects following the same specification or whose configuration changes dynamically. This is a challenging task in the framework of LOTOS, which is better equipped to support the specification of static configurations.

A reflection of the outcome of the synthesis transformation, the compound form, shows that it is not simpler than the distributed form and it does not seem to enhance modularity, at least in any obvious way. The controller units of individual interactors have been 'factored out', and they have been 'divorced' from their corresponding ADUs. Rather than a simplification of a LOTOS behaviour expression, synthesis changes the 'grain' of the architectural description. For example, it may be better to think of the player bar of this example as a single interactor (the compound form) and not as a group of interactors. Whether or not this is the case depends on the purpose and the context of the specification exercise and the designed software architecture. The regrouping of interactors supported by the synthesis transformation does not change the meaning or the complexity of the specification. Rather, it moulds the specification into the form that fits the architectural design of the interface. For example, at the initial stages of an interface design, a single compound interactor describing the interface to Simple PlayerTM may be a more appropriate description, rather than the distributed form corresponding to the graph of Fig. 6. If a higher level description is adopted, many of the details of the specification and the configuration of the interactors are irrelevant. This may be the case for a bottom-up design process, where the compound form is used as a building block for more complex expressions without reference to its internal detail. It is equally the case for top-down design where specifications of pre-defined components are combined, or where the configuration of the interactors has not yet been worked out. In such cases, the need for a more abstract description of interactors is apparent and is supported by the concept of the abstract view.

The practical impact of the synthesis transformation in the design life-cycle of interactive systems has not yet been investigated more than cursorily. At a first glance it seems to support the bottom-up construction of user interface specifications. It is not the intention of this research and it should not be the intention of a representation technique, formal or not, to prescribe a particular approach to interface design. What is required from such a scheme is that it provides the appropriate flexibility for its use in realistic settings. The use of ADC interactor specifications is also consistent with the standard software engineering approach to the refinement of abstract specifications. Standard techniques for the refinement of LOTOS specifications can be used for this purpose. Markopoulos [14] discusses the reverse of the synthesis transformation, the decomposition transformation, which supports the stepwise refinement of ADC interactor specifications. In LOTOS terms, decomposition is a transition from the constraint oriented to the resource oriented specification style [15].

Formal interactor models are a direct analogue of object-based software architectures like PAC [13], ALV [12] and MVC [11] in the context of the formal specification of user interfaces. Contrary to such informal architectures, formal interactor models have to be associated with precisely defined semantics of their compositions. The research reported is a contribution in this direction. The synthesis transformation allows the general structure of the interactor to be maintained throughout the composition of lower level entities. Thus the higher level composition will have the general ADC interactor structure.

The consistent structure which is preserved through the composition of interactors should enhance the comprehensibility of the specifications. Also, it enables the use of generic expressions of properties of interfaces capitalising on the standardised syntactic structure of the interactors. Markopoulos [14] discusses also how ADC interactor specifications support the formal analytical evaluation of interaction dialogues. Some interesting results in the automatic verification of interaction properties are reported by [20] who showed how interactor specifications prompt generic expressions of usability related properties which can be verified automatically by the use of model checking tools. Extending this work, Mezzanotte and Paternó classify interaction properties with respect to if and how their automatic verification is feasible.

The integration of formal interactor models within more psychologically informed design approaches is also an issue of current research. Markopoulos et al. [21] propose a formal framework for testing interface specifications against formal expressions of task models.

It has been argued throughout this paper that the synthesis of interactors to form higher level entities has direct relevance in the context of interface design. The work reported here has addressed, at a theoretical level, some of the issues arising in scaling up interactor specifications. This is on-going work and theoretical and practical implications of the various compositions are still under investigation. The instantiation of the ADC formal interactor model, the use of the libraries of specification components, and the transformation of formal specification are expensive activities. Plans for future work include the development of special purpose tools for constructing and editing ADC specifications which should relieve the specifier from the logistics of these tasks. This research uses general purpose tools which support specification in LOTOS, e.g. the LITE toolset [22] and the CADP toolset [23]. The view advocated with the ADC formal interactor model is that practical benefits for the task of specifying formally user interface software can be reaped from the systematic re-use of specification components, the standardisation of specification styles, and the development of purpose specific tool support.

Acknowledgements

This research is funded by EPSRC grant number GR/K79796. Acknowledgement is due to INRIA for providing the MiniLite and the CAESAR/ALDEBARAN toolsets.

Appendix A.

This appendix introduces a few elements of the LOTOS language which are used in the paper. The reader is referred to Ref. [24] for a full tutorial on the language. 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. Sequencing information about such interactions is given by behaviour expressions. The simplest behaviour expression is just a process instantiation. More complex behaviour expressions are built up by the composition of process instantiations with LOTOS operators. The LOTOS operators used in this paper and their meaning are summarised in Table 2.

LOTOS supports the multi-way synchronisation of processes. This means that any number of processes can synchronise over a gate, participating in a single synchronisation on each interaction on this gate. This feature supports the incremental specification of constraints on the occurrence of events at a particular gate, which gives rise to the constraint oriented style of specification discussed by Vissers *et al.* and used throughout this paper.

A process may specify its participation in an interaction with an action declaration. This can be just the 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, $g?x:\text{integer}$ specifies a set of actions $g\langle v \rangle$ where $\langle v \rangle$ 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. $g!\text{true}$, and B offers any event $g?x:\text{Boolean}$, then the value true is passed from A to B.

The semantics of LOTOS operators are defined formally in terms of labelled transition systems. The equivalences used to compare LOTOS behaviour expressions in the paper are introduced below, adapted from LOTOSPHERE [25]. In the following, the notation

$$s \xrightarrow{\mu} r$$

denotes a transition from a state s to a state r which is labelled by an action μ .

Table 2
The LOTOS process operators used in this paper

Name	Syntax	Meaning
action prefix	$a; B$	a then behaviour B
selection predicate	$[b]- > a; B$	if b then a then B
choice	$A B$	behaviour A or B
interleave	$A B$	A and B are concurrent but do not synchronise with each other
disable	$A[>B$	perform A until it terminates, unless at some point B interrupts A
parallel composition	$A [a, b, \dots] B$	A and B synchronise at gates a, b, ... and interleave on others

Appendix A.0.1. Definition. Strong bisimulation equivalence

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

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

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

Two processes P and S are called strong bisimulation equivalent if there exists a strong bisimulation relation R relating their initial states of their LTS, i.e. $(p_0, s_0) \in R$.

The transition relation $s \Rightarrow 'r$ denotes that a string of observable actions t , effects a transition from a state s of the process to a state r . Actions are qualified as ‘observable’ to differentiate them from silent internal actions which can be the hidden action i or the successful termination event δ . By substituting $s \Rightarrow 'r$ in the place of \rightarrow^μ in the definition above we obtain the weak bisimulation relation and the weak bisimulation equivalence, denoted as $P \approx Q$.

These definitions extend to full LOTOS processes. Two LOTOS behaviour expressions P and Q , with all their free value-identifiers contained $\{x_1, \dots, x_n\}$ are weak (strong) bisimulation equivalent, if all instances $[E_1/x_1, \dots, E_n/x_n]P$ and $[E_1/x_1, \dots, E_n/x_n]Q$ are weak (respectively strong) bisimulation equivalent where E_1, \dots, E_n , are closed value expressions of the same sort as x_1, \dots, x_n respectively.

A *LOTOS context* $C[.]$ is a LOTOS behaviour expression with a formal process parameter denoted by ‘ $[\cdot]$ ’. If $C[.]$ is a context and B is a behaviour expression then $C[B]$ is a behaviour expression that is a result of replacing all ‘ $[\cdot]$ ’ occurrences in $C[.]$ with B . Two LOTOS behaviour expressions B_1 and B_2 are called *weak bisimulation congruent*, denoted $B_1 \cong B_2$, if for all LOTOS contexts $C[.]$, $C[B_1] \approx C[B_2]$.

Appendix A.0.2. Definition. Naive transformation from full LOTOS to basic LOTOS.

This transformation maps a LOTOS specification to a basic LOTOS specification, by mapping each interaction of the original specification to an interaction characterised by the same gate name but is not associated with data values. The definition below of the mapping P from full LOTOS behaviour expressions to basic LOTOS expressions, is adapted from Ref. [25]. The mapping is defined here only for the LOTOS constructs used to specify ADC interactors. In this definition g denotes a gate, e denotes an ACT-ONE value expression, x denotes an ACT-ONE value identifier, s a type sort, B a full LOTOS behaviour expression, Q a LOTOS process identifier.

$$P(g!e; B) = g; P(B)$$

$$P(g?x : s; B) = g; P(B)$$

$$P(B_1 || [G] || B_2) = P(B_1) || [G] || P(B_2)$$

$$P(B_1 [] B_2) = P(B_1) [] P(B_2)$$

$$P(Q[g_1, \dots, g_n](e_1, \dots, e_k)) = Q[g_1, \dots, g_n]$$

$$P(Q[g_1, \dots, g_n](x_1, \dots, x_k)) : = B) = Q[g_1, \dots, g_n] : = P(B).$$

References

- [1] M.D. Harrison, A.J. Dix, A state model of direct manipulation in interactive systems, in: M.D. Harrison, H.W. Thimbleby (Eds.), *Formal methods in human computer interaction*, Cambridge University Press, Cambridge, 1990, pp. 29–151.
- [2] A.J. Dix, *Formal methods for interactive systems*, Academic Press, New York, 1991.
- [3] D.J. Duke, M.D. Harrison. Abstract interaction objects, in: R.J. Hubbold, R. Juan (Eds.), *EURO-GRAPHICS'93, Computer Graphics Forum 12(3)* (1993) 26–36.
- [4] F. Paternó, G.P. Faconti, On the use of LOTOS to describe graphical interaction, in: A. Monk, D. Diaper, M. Harrison (Eds.), *People and Computers VII, Proc. Human Computer Interaction 1992*, Cambridge University Press, 1992, pp. 155–173.
- [5] P. Markopoulos, On the expression of interaction properties within an interactor model, in: P. Palanque, R. Bastide (Eds.), *Design, specification and verification of interactive systems '95*, Springer, Wien, 1995, pp. 294–311.
- [6] ISO, *Information processing systems—open systems interconnection—LOTOS—a formal description technique based on the temporal ordering of observational behaviour*. ISO/IEC 8807, Int. Organisation for Standardisation, Geneva, 1989.
- [7] P. Markopoulos, Case study in the formal specification of the SimplePlayer[™] graphical interface for playing QuickTime[™] movies, using the ADC interactor model, Technical Report 712, Department of Computer Science, Queen Mary and Westfield College, University of London, 1996.
- [8] G.P. Faconti, Towards the concept of interactor, AMODEUS project report, ref. sm/wp8, 1993.
- [9] D.J. Duke, G.P. Faconti, M.D. Harrison, F. Paternó, Unifying views of interactors, in: *Advanced Visual Interfaces '94 Conference Proceedings*, ACM Press, 1994, pp. 143–152.
- [10] B.A. Myers, A new model for handling input, *ACM Transactions on Information Systems* 8(3) (1990) 289–320.
- [11] G.E. Krasner, S.T. Pope, A cookbook for using the model-view-controller user interface paradigm in the Smalltalk-80 system, *Journal of Object Oriented Programming* 1(3) (1988) 26–49.
- [12] R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson, W. Wilner, The Rendezvous architecture and language for constructing multiuser applications, *ACM Transactions on Computer Human Interaction* 1(2) (1994) 81–125.
- [13] J. Coutaz, PAC, an object oriented model for dialog design, in: H.J. Bullinger, B. Shakiel (Eds.), *INTERACT-'87*. Elsevier, North-Holland, 1987, pp. 431–436.
- [14] P. Markopoulos, A compositional model for the formal specification of user interface software, PhD thesis, Queen Mary and Westfield College, University of London, March 1997.
- [15] C.A. Vissers, G. Scollo, M. van Sinderen, E. Brinksmia, Specification styles in distributed systems design and verification, *Theoretical Computer Science* 89 (1991) 179–206.
- [16] F. Paternó, Formal reasoning about dialogue properties with automatic support, BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, *Interacting with Computers*, 9(2).
- [17] R. Guindon, Designing the design process: exploiting opportunistic thoughts, *Human-Computer Interaction* 5 (1990) 305–344.
- [18] R. Milner, *Communication and concurrency*, Prentice Hall, UK, 1989.
- [19] A. Ruid-Delgado, D. Pitt, C. Smythe, A review of object oriented approaches in formal methods, *The Computer Journal* 38(10) (1995) 777–784.
- [20] F. Paternó, Definition of properties of user interfaces using action based temporal logic, in: *Proc. of the 5th Conference in Software Engineering and Knowledge Engineering*, 1993, pp. 314–318.
- [21] P. Markopoulos, P. Johnson, J. Rowson, Formal aspects of task based design, in: J.C. Torres Cantero (Ed.), *Design Specification and Verification of Interactive Systems '97*, Springer (Wien), in press.
- [22] J.A. Mañas, Getting to use the LOTOSphere Integrated Tool Environment (LITE), in: T. Bolognesi, J. van de Lagemaat, C. Vissers (Eds.), *LOTOSphere: Software Development with LOTOS*, Kluwer, Netherlands, 1995, pp. 87–107.
- [23] J.C. Fernandez, H. Caravel, L. Mounier, A. Rasse, C. Rodríguez, J. Sifakis, A toolbox for the verification of LOTOS Programs, 14th Int. Conference on Software Engineering, Melbourne, May 1992.

- [24] T. Bolognesi, E. Brinksma, Introduction to the ISO specification language Lotos, in: P. van Eijk, C. Vissers, M. Diaz (Eds.), *The Formal Description Technique Lotos*, Elsevier, North-Holland, 1989, pp. 23–73.
- [25] LOTOSPHERE, Catalogue of LOTOS correctness preserving transformation, in: T. Bolognesi (Ed.), *Final Deliverable of the LOTOSPHERE ESPRIT Project*, Lo/WP1/T1.2/N0045/Vo3, 1991.