The ADC interactor as a conceptual user interface software architecture

P. Markopoulos
Technical Report

The ADC interactor as a conceptual user interface software architecture

P. Markopoulos

1 Introduction

The Abstraction Display Controller (ADC) interactor model, of [MAR97] has been developed in the formal specification language LOTOS [ISO89]. One aim of [MAR97] and of the ARAMIS\(^1\) project is to develop the concept independently of its formal representation. The architecture model proposed here is a step in that direction which aims primarily to validate the formal model by relating it to an implementation architecture. This report describes how the concepts described by the formal model are mapped to constructs of an object oriented implementation. The ideas are still tentative. This report is an input to current implementation work which will test and improve the ideas outlined hereby.

2 Terminology

An architecture is an abstract description of a class of implementations. This abstraction may concern different levels of abstraction. At a concrete level an implementation environment supports and enforces a common architecture for all the systems implemented with it. At a higher abstraction level an architecture is a model, i.e. an analytical tool which serves the purposes of an analysis. This distinction between model and architecture is not inherent in the representation scheme but refers to the purpose of its use. In general, the purpose of an architecture is constructive while a model’s purpose is descriptive and analytical. In this sense, examples of abstract models are the PIE, the red-PIE models [DIX92] while the MVC [KRA88] and the PAC [COU87] ‘models’ are examples of architectures.

An architecture may be an implementation architecture in which case it is supported by some implementation environment, e.g. the MVC [KRA88] and the ALV [HIL94] architectures. An architecture is called a conceptual architecture when it is simply seen as a conceptual framework, a set of guidelines or heuristics for structuring the implementation of the software, e.g. the PAC model [COU87]] and the Seeheim model [GRE85].

\(^{1}\) EPSRC funded project, grant number GR/K79796
As Cockton argues [COC90]:

"An architecture may embody desirable properties of a user interface into component and interface descriptions. They may package the good ways of decomposing a class of systems (here interactive ones)…"

This report discusses a class of models of user interface software called interactor models. Interactor models can serve both purposes, and thus can help relate results from the study of abstract models of interactive systems (formal or not) with their implementation.

As a scheme for decomposing a system description an architecture can be judged by the following criteria:

- does it help write clear and good code?
- does it help write efficient software?
- does it embody some principles of good interface design?

3 Interactors: the concept and its formal representation

Interactors are abstractions used to model interactive systems. Compared to the earlier abstract models of interactive systems as, e.g. the state-display model [HAR90] and the PIE model [DIX91], interactors are more structured in that they model interactive systems as compositions of independent entities. Facenti [FAC93] 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 system 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 [DUK93] 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 system. In the latter case it is assumed implicitly that there is at least a conceptual separation (cf. [ROS88]) between the user interface system and an underlying application (functional core). The two definitions of the interactor concept and the corresponding formal models proposed by [DUK93] and [PAT92] are not inconsistent, but they have different origins and they are intended primarily for different purposes. The interactor model of [PAT92] is better suited as a formal constructive design notation, while the model of [DUK94] is better suited for analytical use. In particular, the model of [DUK94] is a concrete representation of abstract interactor models which help formalise usability related properties, e.g. the predictability and the observability of an interactive system (cf. [DIX91]).

The term interactor has been used also to refer to implementation constructs [MYE90]. 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 [KRA88] and the ALV [HIL94] architectures support the notion of an interactor as an implementation architecture construct. The PAC [COU87] architecture supports this notion at a conceptual level.

These are examples of 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 architectures 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. This is a property of the ADC formal interactor model, which is outlined in the following section, as was shown in [MAR97]. The following sections attempt to bestow this property on an informal conceptual architecture based on the ADC interactor model.

4 Outline of 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 the functional core and the user. Alternatively, the ADC model applies to very small component of the user interface software in which case the ADC interactor adheres to the definition by Faconti [FAC93] quoted previously. The interface software can be modelled 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. Notionally, 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 to which it is connected.

These state components are encapsulated within a dedicated component whose purpose is to control the access of the environment to these state component. This component is called the Abstraction-Display-Unit (ADU). It defines a public ‘interface’ for accessing the state components. The ADU applies no temporal ordering constraints on the order of its interactions (and therefore on the execution of its methods). Temporal ordering is described in a separate component the controller unit (CU). The ADC interactor is formed by the composition of these two components (figure 1), which means that the access to the ADU is controlled by the CU component. In later sections degenerate special cases of interactors are considered. For example, 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.

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 (figure 1). The vertical sides are dedicated to controller connections. 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.

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 groups and 'localises' interactions with a similar purpose. The gate set of an interactor comprise its 'public interface'. 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 effect on the state parameters of the interactor.

5 Adding Architectural Detail to the ADC interactor

The abstraction and the display can be implemented as instances of objects in the target programming language. The ADU object provides a limited access to the abstraction and the display object through a public interface which it specifies. The interface consists of a set of gates, which for the ADU will evoke corresponding methods, which can be classified according to their role as below:

- An display input (dinp) method computes a new value for the abstraction and display by interpreting data which is input from the display side of the interactor, with respect to the current display and the current abstraction of the interactor.

- An abstraction output (aout) method interprets the current abstraction and produces data which is communicated to the application or to other interactors. In some cases this operation can implement an identity function when the value of the abstraction is sent to other interactors as it is.

- A display output (dout) method provides access to the value of the display state of the interactor.

- An abstract input (ainp) operation updates the abstraction and display states with respect to data received from the abstraction side.
The asymmetry in the handling of the abstraction and the display reflects their different roles in the ADC model. While different 'clients' of the interactor may require a different interpretation of its abstraction, computed through a method \textit{aout}, this is not the case for the display. The ADU outputs its display state without further interpretation/transformation.

There are three important deviations from the formal model:

- Because the formal model does not specify a rendering mechanism it specifies the output interactions explicitly. This means that two display statuses have to be specified for the formal model, the current status and the computed display which will be the next output. The computed display becomes the display status only after the next output. For the implementation architecture this distinction is cumbersome particularly when the display of an interactor is output directly to the screen. For such 'output' interactors the distinction does not need to be maintained. When the display state is recomputed, it should be assumed that it is output immediately. For the higher level interactors, it remains to be seen whether the distinction should be maintained or not. In the latter case the display status could be a variable to which all 'client' interactors have access.

- It is not necessary to maintain the distinction between the data type AD and the ADU. The distinction was necessary in LOTOS where the data type AD defined sorts and operations and the ADU supported in the process algebraic framework of LOTOS the notion of a state parameter and a standardised mapping between interactions and operations. An ADU object may now be equipped with public methods corresponding to the gates of the ADU and which invoke a set of private operations which correspond to those of the data type AD. This requirement can be relaxed in the implementation architecture, as long the concept of a role of a gate is maintained.
The interactors are put together in a LOTOS behaviour expression which specifies the connections between them. Rather than a ‘main program loop’ which implements the composition expression, it is better to encode the connectivity among interactors directly within them. Each interactor invokes the methods of its clients interactors. The ADU need not be aware of the other interactors it receives information from or of its own ‘clients’. This responsibility belongs to the CU.

The CU defines the flow of control and the flow of data. First it describes the temporal ordering of interactions on all the gates of the interactor. The CU is also responsible for maintaining a list of clients for output: it will send the output messages to all the clients of an interactor. The temporal sequencing of interactions is programmed inside the controller unit in what is called the constraints component. Constraints also involve interactions which have no effect on the abstraction or display data which are characterised as controller gates (c for short). When the CU receives an input it invokes the relevant method of the ADU interactor. When it sends some output it must be aware of a list of recipients for this information. This data flow information is described in the data flow component of the CU. Thus the CU encodes part of the behaviour specified by the LOTOS composition expressions and by what was called the logical connectives for the ADC model.

Consider for example a slider (figure 3).

- The display comprises of an elongated rectangular object and a square indicator positioned along the rectangular object.
- The abstraction consists of a bounded value representing, e.g., time.
- The abstraction display unit defines the following methods:

  * Click and Drag are dinp methods which cause the thumb to move to the mouse position and the natural number to be updated accordingly.

  A newOffSet changes the position of the slider, while a newUpperRange will let the value N range between different limits, e.g. 24.
A setTime method output the value N and sends it to the list of recipients for the slider.

- For example it might be required that input is only sent to the clients after an external ok interaction has occurred, e.g. with an ok button. This ‘dialogue’ is specified in the constraints component. In this case, the ok gate is considered as a control gate for the slider interactor (role c).

Examples of interactors can be buttons, menus, palettes, the editable ‘canvas’ of a window, etc. Even abstractions common to many interactors can be modelled as interactors, supporting multiple views of the same object. Windows and dialogue boxes can be modelled as ADC interactors. Usually the window or dialogue box encloses several interactors. The window’s controller will define the composition of interactors, while its presentation should define windowing operations.

The dialogue box is no different than a window, apart from the fact that the dialogue of some parent window is suspended when the dialogue box is active. A controller only interactor may implement this behaviour. Also dialogue boxes have standard behaviours like ok, cancel, apply, etc., which are particular to the style of interaction supported.

In [MAR97] a set of boundary cases for the ADC model were described. It was mentioned already that a controller unit may in itself be considered as an independent interactor (a controller only interactor). Below several uses of this concept are discussed. An interactor that does not have a display could still be implemented as outlined above, just by omitting dinp and dout methods, and correspondingly a display only interactor will not support any aout and ainp methods. In both these cases a CU will need to be implemented.

6 Connecting interactors

In [MAR97] several different ways of connecting interactors are described. Some were directly specified by LOTOS process algebra operators. Others necessitated the introduction of connectives to overcome the synchronicity of LOTOS. To ensure the correspondence of the implementation architecture to the formally specified architecture, all these connections are examined below. They are implemented in the data flow component of the CU. The connections mentioned in [MAR97] are examined in three different categories: those implemented by message passing, those implemented as a constraints component and those implemented in a data flow component.

6.1 Compositions implemented with message passing.

LOTOS operators can help specify complex behaviours by composing ADC interactors. This section examines the meanings that such compositions might have and their implementation in an asynchronous object based language. The notion of
the role of a gate was introduced in section 3. This may be the role \( d_{inp}, d_{out}, a_{inp}, a_{out}, \) or \( e \). A gate may have different roles for each of the two interactors. A different type of connection is obtained for each combination of roles, e.g. \((d_{inp}, d_{out})\). This discussion does not distinguish between interactors \( \text{ADC}_A \) and \( \text{ADC}_B \), so a connection type is symmetric, i.e. the same type of connection corresponds to the combinations \((d_{inp}, d_{out})\) and \((d_{out}, d_{inp})\). The parallel composition operator of LOTOS [[G]] may specify no connection (pure interleaving) or several of the connection types (a)-(l) below (see also figure 4):

(a) Connection type \((d_{inp}, d_{inp})\). 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 \((a_{inp}, d_{inp})\). 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 \((d_{out}, d_{inp})\). 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 \((a_{out}, d_{inp})\). Data is sent from the abstraction side of A to the display side of B. For example, the thumb interactor of figure 3 should send its result to an interactor modelling the whole slider.
(e) Connection type \((c, \text{dinp})\). 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 \(\text{dinp}\) only when interactor A synchronises with it, which will be only when the appropriate key is pressed.

(f) Connection type \((\text{ainp}, \text{ainp})\). 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 \((\text{dout, ainp})\). 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.

(h) Connection type \((\text{aout, ainp})\). A value is communicated from A to B. For example, the slider of figure 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 \((c, \text{ainp}), (c, \text{dout}), (c, \text{aout}), \) 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. Figure 5 illustrates those omitted. These connection types concern pairs of interactors which synchronise over common output gates and thus introducing the possibility of a deadlock when the two interactors attempt to output a different value.
6.2 Connections implemented as a separate constraints component

The connections discussed in this section concern the dynamic operators of LOTOS, choice and disable.

The choice operator of LOTOS (denoted by $[]$) helps 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.

The disable operator of LOTOS (denoted by $\triangleright$) helps 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.

In both these cases, the most modular approach to the implementation of the software is to encode the choice or the dialogue in a separable controller unit that coordinates the two dependent ADC interactors.

6.3 Connections specified in the data flow component.

The term logical connectives was used in [MAR97] to describe a set of behaviour expressions that support the communication of data between interactors. LOTOS is a synchronous language so sometimes the communication of data between interactors may introduce unwanted dependencies. For example, considerations of modularity suggest that the sending interactor should not be ‘aware’ of a list of recipient interactors. Logical connectives were introduced with the aim to factor out such communication dependencies. Some are processes that buffer information between producers and consumers. In other cases, the required communication scheme is supported directly as a LOTOS process algebra operator.

Connections between ADC interactors represent data flow or pure synchronisation. In the case where data is communicated the ADC model distinguishes the producer from the consumer of the data. Communication schemes may be classified by the number of the producers and the consumers of data, and by whether they should all receive the data (AND) or just one of them should (XOR). The possible connections between ADC interactors are presented in table 1 along with their visual representation. These may be implemented as different classes of data flow components.
Table 1. Classification of data flow components supported by the formal model.

The set of expressions of table 1 are the minimum necessary general-purpose ‘logical connectives’ needed to compose the formally specified interactors. Without their introduction an ADC interactor specification would have to be modified depending on the context of its use. They can be implemented as variations of some generic event queue or buffering mechanism and in the simplest cases directly by message passing.

7 Some comments on the proposed architecture

[COC90] argues in favour of user interface architectures and proposes four dimensions for their classification. These are:

- **Orientation.** It concerns the levels of system decomposition which an architecture provides directly. The design and implementation of the system starts from that level and determines the orientation in which the system will be constructed: top down, bottom up, middle out.

- **Topology.** This dimension concerns the graph structure for control and data flow between components: e.g. pipeline, hierarchic, by-pass pipeline, etc.

- **Component Provision.** This dimension concerns the provision of the basic components. For example, they could be abstract descriptions, or low level building blocks, or protocol descriptions etc.

- **Control and Data Propagation.** This dimension concerns the nature of the inter-relationship between components: e.g. client-server, master-slave, etc.

Supporting his own architecture model Cockton [COC90] argues in favour of a combination of a top down and a middle-out orientation, a cross link topology, user-centred component provision, and the automatic support for control and data
propagation, e.g. through a constraint satisfaction system. In a top-down architecture a set of very abstract components are identified and they are configured or during design. In a middle-out architecture the higher levels of the software structure reflect a specific design rather than a generic application-independent architecture. The designer builds up, rather than fills out, an interactive system. Some basic components can be combined in prescribed ways until the top levels of the architecture are reached. However, not all the higher level components need to be constructed by the composition of basic components. A hierarchical structure which supports links from generic top level components to application-specific components is called a cross-link hierarchy. User centred component provision means that the basic building blocks provided are only those experienced by the user. Alternative strategies for component provision are: functional if the components match application independent functions of a system and information flow if the architectures do not match anything perceived by the user, but model the internal (to the software) information flow.

The implementation of the ADC interactor model described above, can be characterised along these dimensions as follows:

- It is a middle out architecture. It defines a generic structure of its basic components and standard ways of putting them together. Components can be added and described at any level of abstraction in a uniform fashion. Also, a basic set of interactors corresponding to interaction toolkit components may be defined and re-used.

- A directed graph structure describes the component interrelation. The ADC model does not prescribe a particular structure for this graph, which the implementer/designer is free to configure according to the problem in hand.

- Both user-centred component provision and information flow component provision is supported. Interactors which do posses a display component are notionally experienced by the user (even through other interactors), so component provision is user centred. However, the composition graph models information flow (data and control) which do not concern the interaction as experienced by the user.

- Flow of control and data is explicitly coded in the controller units of the interactors, implementing a master-slave relation between the interactor evoking a method and the one implementing it.

8 References


