

# Refinement of the PAC model for the component-based design and specification of television based interfaces

Panos Markopoulos, Paul Shrubsole and John de Vet  
Philips Research Eindhoven  
The Netherlands  
{panos, psh, devet}@natlab.research.philips.com

**Abstract.** Componentisation of software promises to deliver cost efficiency that has not been achieved through object orientation [19]. PAC [5] is a popular conceptual architecture for structuring user interface software in an object oriented fashion. This paper reports our experience of adapting and refining PAC as a component architecture in the context of consumer electronics, and On-screen Displays in particular. The paper describes a structured scheme for the specification of user interface software components, distinguishing ‘look’ and ‘feel’ specific components, and fostering their modular development and re-use.

**Keywords.** PAC, user interface, software architecture, on-screen display, specification.

## 1 Introduction

This paper discusses a component architecture and a specification scheme for On-Screen Displays (OSD), i.e. graphical user interfaces for consumer electronics devices. The OSD may be supported, e.g., by a ‘Set-Top Box’ which is a device that displays, on an analogue television-set, a graphical user interface, e.g., for controlling its own signal processing functionality, a video recorder or electronic-program-guide (EPG) information provided by broadcasters, etc.

The component architecture discussed is an application and refinement of the Presentation Abstraction Control (PAC) model of graphical user interface software [5]. The architecture addresses the following requirements, which are typical for the consumer electronics domain:

- The user interfaces specified will be supported on a variety of platforms with different graphical output abilities: e.g., text only output, pixel based graphics, and different refresh rates for animation. The same ‘feel’ should be supported on all platforms, although the ‘look’ must adapt to the target platform. This may range from text only to animated, 3-D graphics. Thus, ‘look’ and ‘feel’ specific components should be specified and implemented independently.
- User interfaces are typically made for product families of possibly hundreds of variants supporting different languages and features. To support product variations at the architectural level, components should be treated as binary entities (i.e. not as source code) which can be composed by third parties relying only on their interface descriptions [19]. The composition of components relies

on the specification of component interfaces (the concept is explained in more detail below).

- There are limited computing resources available for graphical user interfaces. Low to middle end products have processing abilities comparable to those of personal computers 10-12 years ago. High-end products can match current (low-end) personal computers in processing power, but they can only use 10-20% of their processing power to support graphical interaction. Most processing power is dedicated to signal processing.
- Implementations are mostly in C and are not object oriented. To avoid memory and computational overheads binding of messages to methods (or more generally procedure names to procedures) is static, as opposed to run-time binding which characterises object oriented systems.
- For current graphical user interfaces for OSD, the user input device is a remote control (rather than, e.g., a mouse) and there is no 'free moving cursor' supported. The implication is that the user interface architecture can be simpler than that required for direct manipulation interfaces. The focus of user actions changes through a 'jumping highlight' between large 'blocks' on the screen, which are accessed through constrained remote control events. Graphical objects are not dragged and dropped and no 'fine grain semantic feedback' is required. The latter requirements have been quite influential in shaping user interface software architectures for desktop computers (including the PAC model), but currently they are not necessary in the OSD domain.
- Early descriptions of the PAC model, e.g., [3,5], can be seen as a set of guidelines for structuring applications. To render the model operational, a specification technique is required that maps in a straightforward manner to implementation constructs.
- OSD interaction designers are mostly non-software engineers. Their specifications should be transparent to implementation concerns but it should be possible to relate them to specifications used by software designers. The specification of software components is therefore associated with a 'designer oriented' specification of the design concept proposed. This was based on the Interaction Styles template [9] which is discussed briefly and exemplified in the following sections.

The remainder of the paper discusses an architectural model that is shaped to address these requirements. Section 2 discusses related work. Section 3 presents the architectural model and the specification scheme derived. Section 4 discusses an example. Section 5 discusses the contributions and limitations of this work and current work.

## **2 Related Work**

The refinement of PAC discussed here is an architectural model which can be used *prescriptively* to design, specify and implement interactive software components, and has been applied *descriptively*, to classify software components that support advanced graphics (advanced refers here to the state-of-the-art OSD). The intended users of this model are software engineers who design and implement graphical interface software.

The aim is to encourage the modular development of software components, their inventorisation and systematic re-use.

The requirement for separating dialogue and functionality related components from presentation components is well served by the PAC model. However, this model needs to be articulated more clearly in an architectural framework (as argued by Coutaz [7]). This paper undertakes this effort within the limiting requirements of the consumer electronics domain outlined in section 1. Related work is [5], which provides a thorough tutorial description of the PAC model as a design pattern and provides guidance for its object oriented implementation. Also the AMF model [20] is a rendition of PAC as a design pattern, that focuses on the composition of PAC agents.

Earlier formal renditions of the PAC model have used formal modelling languages such as Z to capture the essence of the model but were not intended for constructive use, e.g., [2, 11]. The intention of the current approach is not towards analysis of the model itself or of systems built following the model, but to provide a blueprint for the implementation of the actual software and to guide the componentisation of the software produced.

The interaction techniques also need to be described independently of their implementation. Such a description is called here *designer oriented*. It uses the interaction styles template [9] used within Philips to specify user interfaces. The interaction styles templates is a variant of the User Action Notation [10]. It is simpler than UAN as it only describes single interactions steps. (In OSD systems interaction is via the remote control, individual interaction steps should be intuitively designed and specified and long interaction sequences are unlikely). The effect of each action is described through illustrations showing the display before and after the user action.

[15] compares the specification of user interface software from a software architecture viewpoint and a user, task-oriented viewpoint. The formalisms compared were both informal specifications structured in tables to reflect elements of the ADC formal interactor model [14] and the User Action Notation [10]. Although the comparison reflected the experience gained from a small case study, it shows that architectural and user action oriented descriptions provide complementary viewpoints at a similar level of abstraction. In the approach reported here, interaction styles specifications were paired up with component specifications, thus providing complementary descriptions of the user interface software. This practice is illustrated in the example of section 4.

The specification scheme presented is not formal. Formal specifications of user interface architecture are well served by interactor models as documented in earlier events of the DSV-IS series. From an architectural viewpoint, a criticism that applies to formal interactor models is that they do not help the developer determine the actual software structure: they rely on semantic constructs of the specification language used which do not necessarily map to the actual programming language constructs used. For example, LOTOS [12] used in [14], supports a synchronous communication of events. In the context of the study reported hereby, software components communicate with method calls. The exact nature of the method invocation mechanism in the target implementation platform is normally abstracted away in a

typical formal specification but is an important concern for the structure and the content of the component architecture.

A range of communication schemes may be envisaged to model component interactions, e.g., streams of data, data sharing, constraints between the states of two objects, message sending, etc. The most modular approach requires the asynchronous selective broadcast of events which *implicitly invoke* methods inside objects [18]. Unfortunately, implicit invocation incurs prohibitive computational costs for current consumer electronics platforms. For this reason we assume that components comprising the user interface communicate with each other and with the application directly via method calls. Asynchronous user events are assumed to be serialised by some input manager. Processing individual user events can thus proceed in a serial fashion: all processing caused by one event must be completed before another one is processed. Exceptions to this rule may be necessary as a result of the asynchronous nature of user input, e.g., reversing the direction of movement within an animated menu structure. Such behaviour is encoded within individual components.

### **3 A component based architecture for interactive software for the next generation OSD**

#### **3.1 Scope of the model**

The scope of the model discussed spans the presentation and interaction toolkit layers of the Slinky/Arch reference model [4]. Device drivers and input libraries provide device independent interpretations of user input to the components discussed here (e.g., a typical interpretation of a button press on the remote control would be “North” or “South”). Device drivers map abstract descriptions of output to invocations of graphics software (e.g., function calls to a graphics library). We assume the existence of an input manager handling the dispatch and serialisation of input events and a screen manager which merges the output of interactive objects. Thus, we model the display at an object by object basis rather than how the displays of individual interactive objects are merged together at runtime. This (quite standard) approach is consistent with the macroscopic software architecture for our target platforms.

#### **3.2 ‘Look’ and ‘Feel’ specific components**

Interactors are characterized in terms of their presentation, their abstract functionality and their reactive behaviour. Figure 1, illustrates the decomposition of an interactor into a *behaviour component* and a *presentation component*. The intention of this division is that the presentation component captures the ‘look’ of the interactor while the ‘feel’ is defined by the behaviour component. This division encourages their independent design and implementation and allows for multiple presentations to be connected to the same behaviour component as long as they have consistent interfaces. The notion of a *behaviour component* is introduced to describe a software component encapsulating the display-independent intrinsic behaviour of an interactor that supports a *basic user task*. By basic user task we mean a simple and recurring operation that the user performs and which is generic over the application-domain

studied - in the present context TV-based and VCR-based graphical user interfaces. Examples of basic user tasks are: 'input a string' (e.g., a programme or channel name), 'set a value' (e.g., the volume) and 'choose among options' (e.g., selecting a channel). The display dependent data and functionality of an interactor are supported by a *presentation component*. This definition de-couples the notion of a behaviour component from the description of the *presentation* which visualises the data of the behaviour component. The presentation component implements a *design concept* which is particular to the behaviour component.

As an example, consider the task of selecting one of a small number of options supported by a vertical linear on-screen menu. To describe the corresponding design concept it is assumed that generalised "next" and "previous" interactions support navigation within the menu. The design concept directly concerns the visualisation of the menu and the "item in focus". How one should implement this idea is of no importance to the design concept. On the other hand, the behaviour component definition for (say) a pull-down menu, determines what data is held by the menu (e.g., a list of options), how the input received (e.g., a button press on the remote control) is interpreted to make a selection. The presentation component details how the information held by the behaviour component relates to the one on the display (e.g., that there is only one item in focus), it visualises this data and controls the animations for pulling down the menu, or for scrolling through its items.

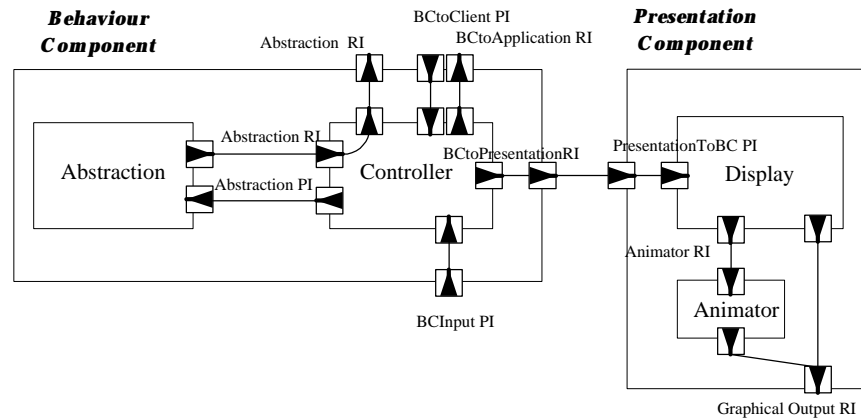
Generally the separation of behaviour and presentation in graphical interaction is problematic: the interpretation of user input is normally display-based (see ., e.g., [13]). However, in OSD interfaces, there is no pointing device used and in the majority of cases the separation of behaviour and presentation is feasible (and desirable for the reasons mentioned in section 1).

### 3.3 Components and interfaces

Components and their interconnections are specified using elements of the graphical notation introduced with the Koala system [17]. Koala is a tool that supports the 'gluing' of components, implemented in C, at configuration time (i.e. before compilation). Given the source code for components and glue modules, which bind components together, and a specification of the composition structure for putting these components together, Koala matches procedure calls to their definitions, taking care of issues such as diversity between members of a product family and optimises the source code. The present approach does not use the Koala system as such. Only the architectural description language of Koala is used to take advantage of the familiarity of Philips staff with this formalism. The intention is to support a similar graphical formalism for the composition and the deployment of pre-fabricated binary components. Some elements of the Koala notation are summarised below.

Blocks represent components (see figure 1). Squares containing triangles represent interfaces. Interface squares are placed on the boundaries of component boxes. We note the following conventions concerning interfaces and their specification in the remainder of this document:

- An interface makes some part of the encapsulated functionality of a component available to its environment.



**Fig. 1.** The decomposition of interactors to a 'behaviour' and a 'presentation' component.

- An interface will be described as a set of method calls, where each method is described by: the name of the method, the type of its arguments in round brackets, followed by the return type, e.g., *add(integer, integer):integer*. Parameter and return types may be omitted if there are none associated with the method call. In the specifications that follow we rely on the naming of these parameters and functions to convey their meaning – i.e. the types mentioned are not defined further.
- An inward pointing triangle represents a *provides interface (PI)*, which means that the component can provide a service to its environment.
- An outward pointing triangle represents a *requires interface (RI)* which models functionality the component requires from the environment.
- Connections between interfaces are restricted to directed connections from one or more requires interfaces to a single provides interface.
- Components can be nested to model compound components (see for example figure 1). Where interface connections cross boundaries of nested components we assign a unique name for the interface so as not to clutter the description (see for example the *Abstraction RI* interface in figure 1).

Note that the direction of an interface (provides or requires) is not directly associated with the data flow, but expresses dependency relationships between software components, i.e. the direction of method calls. For example, figure 1 stipulates that in order to operate successfully an interactor requires some graphical output functionality. Similarly the presentation provides some functionality to the behaviour component, etc.

Abstraction	Encapsulated data and its invariants.
Abstraction RI	Operations to initialise/modify the abstraction with data from the application or other interactors.
Abstraction PI	Operations upon the encapsulated data. They describe functionality that the behaviour component provides to its environment through the controller.
BCInput PI	The set of method-calls implementing input to the interactor. Usually reserved for user input which the behaviour component must interpret or for user initiated interrupts.
BCtoClient PI	Functionality the interactor provides to its environment. The receiver of this information has the initiative for calling the relevant method.
BCtoClient RI	Functionality the interactor provides to its environment. The BC keeps the initiative for calling the required methods to convey its results to other components.
BCtoPresentation RI	Method calls to visualise the abstraction data in the display and to control the display and its animators.
Control	Sequencing constraints implemented by the controller.

**Table 1.** Structured specification of a behaviour component.

### 3.4 Behaviour component: internal structure and specification

The behaviour component comprises of an abstraction component and a controller component. The controller component is responsible for handling user and system events to interrupt the currently executed method. User interrupts override or cancel the current control task, e.g., when the user reverses the current direction of movement within an animated menu. Interrupt messages should be translated to the presentation component in the form of method calls to affect the relevant aspects of the display, e.g., pausing or canceling an icon based animation.

The controller localises control, i.e. some temporal sequencing on its communications with its environment. As mentioned already, the provides and requires relations describe static dependencies between software components. The control specification captures dynamic behaviour. In general, the behaviour component drives the interaction with the presentation, while no assumptions are made with respect to whether it controls an application or vice versa (mixed internal/external control of the interface by the application is possible).

- *BCinput PI* models input in the form of method calls originating from the user through logical input devices or through other interactors mediating with the user. This interface includes methods for interrupt handling.
- The *Abstraction* is modelled as an abstract data type (ADT) and may be implemented as a module. The controller can access the abstraction state through its provides interface (*Abstraction PI*), which is a set of query and modification operations.

Display	Data and its visualisation and where applicable correspondence of abstraction to display features.
PresentationToBC PI	Provides interface to the behaviour component: a set of methods and their arguments.
Graphical Output RI	Requires interface for the graphical output (i.e. some subset of the target GDI called from within the presentation component).
Animator RI	A requires interface for the animator components (which are not specified explicitly in this paper).

**Table 2.** Structured specification of a presentation component

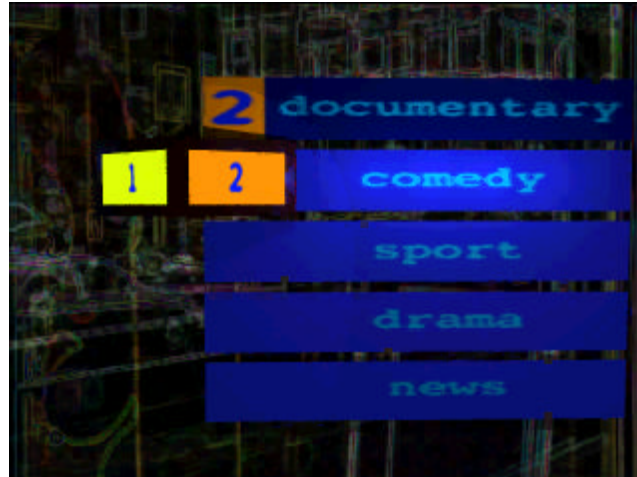
- The abstraction receives data for its initialisation from the application through the controller and specifically through the *Abstraction RI* interface.
- *BCtoClient PI* and *BCtoClient RI* communicate the result of the interaction to the application. If the behaviour component in question has the initiative for outputting some data it will do so through the RI to the PI of some other behaviour component or the application. If not, it will provide this data through its RI.
- The behaviour component reads and modifies data of the presentation component through *BCtoPresentation RI*. In doing so it ensures that changes are propagated from the abstraction to the presentation and vice versa. Also, it includes methods for interrupting the presentation.

A structured scheme for the specification of behaviour components is shown in Table 1. The first three rows define the abstraction data encapsulated by the behaviour component: the data it encapsulates, the operations to initialise/modify it and inquiry operators for reading its value. The interface to the presentation is a requires interface: the behaviour component requires some functionality for its visualisation. The control specifies the dynamic behaviour of the interactor, i.e. how it maps user input to invocations of functionality in the presentation and application.

### 3.5 Presentation component: internal structure and specification

The presentation is broken down into the *display component* and one or more *animator components* (see fig. 1). The display component encapsulates data and behaviour related to the presentation of data on the screen. One or more animator components can be invoked by the display, visualising some transient effect between display states. The display component controls the animators, i.e. it triggers them, it can interrupt them, reverse them, etc. Presentation components can be specified as in Table 2.

- *Display data* describes the data which is visualised by the presentation component, its visualisation and, where appropriate, the correspondence of features of the visualisation to the abstraction of the behaviour component.
- *PresentationToBC PI* is a set of methods that the presentation provides to the behaviour component. It serves the communication of data to and from the behaviour component when it is connected to the *BCtoPresentation RI* of the latter and the interruption of current animations by the behaviour component.



**Fig. 2.** The wheel selector design concept. Screen shot from non-interactive prototype.

- *Graphical Output RI* is the set of methods required for graphical output. These methods are left unspecified in the specifications of this document, to avoid unnecessary implementation bias in the absence of a generic graphical device interface for the candidate target platforms (e.g., the G+4 UIMS [1], Windows CE). The specification and implementation of a generic graphical device interface for all the target platforms is currently developed.
- *Animator RI* describes the interfaces of the display to animator components in terms of the start and the end state of the animation. States for the animation can be described by, e.g., some spatial/geometrical attribute of the display, the frame number in a sequence of frames/bitmaps.

At any moment an animator component maintains (at least) three variables: its start state, its current state (alias 'ist' state) and its target state (alias 'soll' state). In order to ensure a flexible user interface architecture animators are interruptible. This allows the user to cancel, or halt animators so as to allow other interactors to take precedence. It is the responsibility of the animator component to deal with interrupts: it may speed up the animation, it may stop it instantly or it may choose to reverse the animation by recursively instantiating itself with its *ist* state as a *start* parameter and its original *start* state as a *soll* parameter. In the tabular specifications of this paper we do not specify the animator components as they reflect detailed implementation concerns.

## 4 Example: the Wheel Selector

### 4.1 The design concept and the components to support it

As an example of the component architecture presented, this section discusses the architecture and the specification of software components to implement the 'wheel selector' design concept. Figure 2 is a screen-shot from the original designers' non-

Pre	User Action	System Reaction	Post																																																
<table border="1"> <tr><td>6</td><td>7</td><td>News</td></tr> <tr><td></td><td></td><td>Sports</td></tr> <tr><td></td><td></td><td>Music</td></tr> <tr><td></td><td></td><td>Films</td></tr> <tr><td></td><td></td><td>Documentary</td></tr> <tr><td></td><td></td><td>Talk Show</td></tr> </table>	6	7	News			Sports			Music			Films			Documentary			Talk Show	OK	Set currently selected variable to the value on the front of the wheel.	<table border="1"> <tr><td>6</td><td>7</td><td>News</td><td>7</td></tr> <tr><td></td><td></td><td>Sports</td><td></td></tr> <tr><td></td><td></td><td>Music</td><td></td></tr> <tr><td></td><td></td><td>Films</td><td></td></tr> <tr><td></td><td></td><td>Documentary</td><td></td></tr> <tr><td></td><td></td><td>Talk Show</td><td></td></tr> </table>	6	7	News	7			Sports				Music				Films				Documentary				Talk Show							
6	7	News																																																	
		Sports																																																	
		Music																																																	
		Films																																																	
		Documentary																																																	
		Talk Show																																																	
6	7	News	7																																																
		Sports																																																	
		Music																																																	
		Films																																																	
		Documentary																																																	
		Talk Show																																																	
<table border="1"> <tr><td>6</td><td>7</td><td>News</td><td>7</td></tr> <tr><td></td><td></td><td>Sports</td><td>6</td></tr> <tr><td></td><td></td><td>Music</td><td>5</td></tr> <tr><td></td><td></td><td>Films</td><td>4</td></tr> <tr><td></td><td></td><td>Documentary</td><td>3</td></tr> <tr><td></td><td></td><td>Talk Show</td><td>5</td></tr> </table>	6	7	News	7			Sports	6			Music	5			Films	4			Documentary	3			Talk Show	5	S (similar for N)	Move the wheel down (or up) to the next (or previous) variable on the list.	<table border="1"> <tr><td>6</td><td>7</td><td>News</td><td>7</td></tr> <tr><td></td><td></td><td>Sports</td><td>6</td></tr> <tr><td></td><td></td><td>Music</td><td>5</td></tr> <tr><td></td><td></td><td>Films</td><td>4</td></tr> <tr><td></td><td></td><td>Documentary</td><td>3</td></tr> <tr><td></td><td></td><td>Talk Show</td><td>5</td></tr> </table>	6	7	News	7			Sports	6			Music	5			Films	4			Documentary	3			Talk Show	5
6	7	News	7																																																
		Sports	6																																																
		Music	5																																																
		Films	4																																																
		Documentary	3																																																
		Talk Show	5																																																
6	7	News	7																																																
		Sports	6																																																
		Music	5																																																
		Films	4																																																
		Documentary	3																																																
		Talk Show	5																																																
<table border="1"> <tr><td>6</td><td>7</td><td>News</td><td>7</td></tr> <tr><td></td><td></td><td>Sports</td><td>6</td></tr> <tr><td></td><td></td><td>Music</td><td>5</td></tr> <tr><td></td><td></td><td>Films</td><td>4</td></tr> <tr><td></td><td></td><td>Documentary</td><td>3</td></tr> <tr><td></td><td></td><td>Talk Show</td><td>5</td></tr> </table>	6	7	News	7			Sports	6			Music	5			Films	4			Documentary	3			Talk Show	5	E (similar for W)	Turn the wheel bringing the next (or previous) value to the front	<table border="1"> <tr><td>7</td><td>8</td><td>News</td><td>7</td></tr> <tr><td></td><td></td><td>Sports</td><td>6</td></tr> <tr><td></td><td></td><td>Music</td><td>5</td></tr> <tr><td></td><td></td><td>Films</td><td>4</td></tr> <tr><td></td><td></td><td>Documentary</td><td>3</td></tr> <tr><td></td><td></td><td>Talk Show</td><td>5</td></tr> </table>	7	8	News	7			Sports	6			Music	5			Films	4			Documentary	3			Talk Show	5
6	7	News	7																																																
		Sports	6																																																
		Music	5																																																
		Films	4																																																
		Documentary	3																																																
		Talk Show	5																																																
7	8	News	7																																																
		Sports	6																																																
		Music	5																																																
		Films	4																																																
		Documentary	3																																																
		Talk Show	5																																																

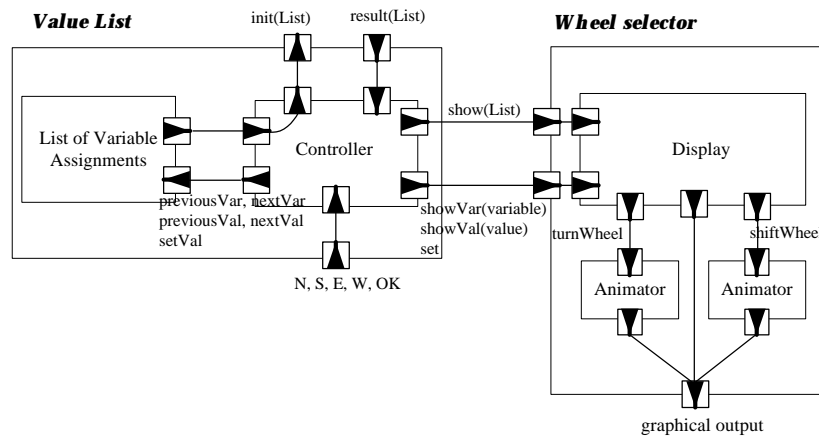
**Table 3.** Specification of the wheel-selector design concept using the interaction styles template.

interactive prototype. The general idea is that the user can construct a ‘profile’ of television-programme-preferences by assigning values to each category of programmes displayed. The user can navigate up and down the programme categories and can turn the wheel left and right to change the value assigned to a particular category. The interaction with this component is specified in Table 3 using the interaction styles template of [9].

Each row of the table represents a single interaction step. A row is read left to right, but contrary to UAN [10], top-down ordering of the rows does not reflect sequence of user actions. The content of the display before and after the user action is shown on the Pre- and Post- columns of the template, while the system reaction corresponds to the ‘connection to computation’ column of UAN. This template form has been successfully applied by non software engineers in Philips for documenting and standardising interaction designs.

Figure 3 illustrates a composition of software components implementing the wheel-selector design concept. The *Value List* behaviour component holds a copy of the list of variables and their values (see specification of Table 4). *Value List* is independent of the presentation (whether for example the values are displayed on a wheel or on an array, etc). The list is initialised (say by the application) with *init(List)*.

Remote control events *N*, *S*, *E*, and *W* are mapped by the controller to invocations of *previousVar*, *nextVar*, *previousVal* and *nextVal* respectively. Event *OK* invokes the



**Fig 3.** The architecture of the selector wheel interactor.

method *setVal* which assigns the *currentVar* to the *currentVal*. Method *result* returns the list of variables and their values.

#### 4.2 Component weight and presentation independence

Supporting multiple views for the same abstraction data may lead to an excessively complex control component. This problem can be overcome by using the Observer design pattern [8] (supported by the MVC model [13]). Using PAC the observer pattern can be supported by hierarchical composition: a single behaviour component (an abstraction-controller pair) with many dependent controller-presentation pairs. The controllers would handle change notification and invoke updates of the dependents.

The approach adopted supports the requirement for presentation independence in nearly all cases except where the semantics of the behaviour component are *solely* dictated by its visualisation. An example of this is text input via a virtual keyboard, where the meaning of user actions for text entry is dictated solely by the visualisation of the character set on the screen (as a keyboard, sheets of characters, wheel selectors, etc.). Another example (which has always been outside the scope of the present work) is direct manipulation user interfaces found currently on desktop computers.

There is a danger that if we attempt to rid the behaviour component of any 'presentation bias' it may become contrived and of trivial size. Presentation independence and display mediated interaction set conflicting requirements for componentisation. The trade-off suggested with the component model discussed above is most appropriate for OSDs of consumer electronics where input is achieved via a remote control. In these cases the user navigates with a small set of buttons between interactive objects of significant size.

Abstraction	A list of variable-value pairs (i.e. a list of variable assignments). A pointer to the current variable and the current value is maintained.
Abstraction RI	<i>Init(list)</i> Operations to set the abstraction list of variable – value pairs with application data.
Abstraction PI	<i>NextVar</i> , <i>prevVar</i> , <i>nextVal</i> , <i>prevVal</i> support navigation with the list data structure. <i>getList</i> returns the current list. <i>setVal</i> sets the current variable to the current value.
BBInputPI	<i>N</i> , <i>S</i> , <i>E</i> , <i>W</i> and <i>OK</i> .
BBtoClient PI	<i>Result(list)</i> The result of the interaction is a list of variable assignments which can be read by the application or higher level components.
BBtoPresentation RI	<i>Show(list)</i> Sends list to the presentation component (the wheel selector). <i>ShowVar(Var)</i> sends the current variable to the presentation component. <i>ShowValue(Value)</i> sends the current value to the presentation component. <i>Set</i> tells the presentation component that the current variable is set to the current value.
Dialogue	<i>N</i> , <i>S</i> invoke the <i>nextVar</i> and <i>prevVar</i> methods of the abstraction and subsequently the <i>showVar</i> . <i>E</i> , <i>W</i> invoke the <i>nextVal</i> and <i>prevVal</i> methods and then the <i>showVal</i> method. <i>OK</i> invokes in order the methods: <i>setVal</i> , <i>set</i> , and <i>result(getList)</i> .

**Table 4.** The value list behaviour component.

### 4.3 Implementation

A selection of design concepts using advanced graphics for OSD, were implemented on an Windows NT based PC system using Win32 GDI, DirectX and OpenGL for the visualisation components. The choice of prototyping platform was based on the grounds of flexibility and performance. As the infrastructure for using the component model is not yet in place, component composition was achieved using the Microsoft COM model [16]. Microsoft COM is a de facto industry standard component model, which facilitates run-time instantiation and binding of components along with extensive support for 3D graphics. Central differences with the Koala model for component composition [17] discussed earlier are:

- COM specifies only *provides* interfaces and not *requires* interfaces, and
- COM components are binary entities rather than units of source code.

The issues of porting user interface components from the PC to platforms for Philips consumer electronics (WinCE, MG and G+4 [1] based systems, particularly with respect to 3D and general performance/resource issues, is under investigation. A few

Display	<p><i>list</i>: The array of variables manipulated. A segment of this list is displayed at any time.</p> <p><i>Wheel</i>: A circular list of values (the range of values for all the variables in the list). The list is rendered as a 3D hexa-hedron, placed next to the current variable and whose front face always shows the current value.</p>
PresentationToBC PI	<p><i>Show(List)</i> initialises the display data with the list to display.</p> <p><i>ShowVar(var)</i> updates the current variable.</p> <p><i>ShowVal (val)</i> updates the current value.</p> <p><i>Set</i> shows feedback of the assignment of the current variable to the current value.</p>
Graphical Output RI	Methods for compressing/blitting bitmap resources.
Animator RI	<p><i>TurnWheel</i>. Turns the wheel <math>\pm 60</math> degrees at a time, depending on which is the new current value.</p> <p><i>ShiftWheel</i>. Shifts the wheel up and down depending on the relative position of the current variable.</p>

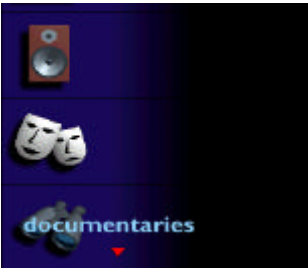


**Table 5.** The wheel selector presentation component.

components for text entry have been ported to G+4 [1] providing a relatively advanced appearance within the resource limitations and graphics capabilities of G+4. Table 6 summarises implementation issues regarding components to support the task of creating a preference profile. Preference profiles are set up by television viewers with the purpose to filter Electronic Program Guide (EPG) information about television broadcasts. The components implemented have been created as both stand-alone demonstrations and as ActiveX control widgets for re-use within authoring environments such as Visual Basic and web authoring tools.

## 5 Discussion

The architecture presented in this paper is a refinement of the PAC [5] model. Contrary to PAC, this architecture does not prescribe a pattern for the composition of interactors comparable to the tree hierarchy supported by PAC. A PAC-like hierarchical composition of interactors is consistent with the present architecture but it is not the only possible way of component connection. The configuration of interactor compositions is currently under investigation.

The architectural model presented in this paper is more specific than PAC regarding the connections between the components, and the P,A,C constituents of PAC agents. It defines the interfaces between behaviour component, presentation and their environment, essentially 'componentising' the PAC model. This refinement is necessary to make the model operational and, also, it ensures the independence of the behaviour component from its presentation component, which was one of the primary aims of the modelling effort.

Component	Visualisation	Visualisation Implementation details
Image List Scroller		Basic 2D GDI calls to perform bitmap blitting (bit block transfers) where the number of image items is too great to display at once. The semantic choice of bitmaps (based on resource identifiers) is determined by the scrolling direction of the user.
Polygon Twister		Real-time 3D animation using OpenGL : 3D implementation allows for much greater flexibility with little coding overhead for alternative visualisations and animations. This comes at the expense of computation. Pre-stored bitmaps with image stretching are an alternative means for visualisation.
Wheel Selector (TV profiler– a compound component)		Combination of above components: The semantics of the image list in this case pertain to categories of TV programs whilst the Polygon Twister refers to the desired rating of the category. A Record of the current preferences is displayed in another image list (shown on the right).

**Table 6.** Software components for creating a preference profile.

PAC was introduced with the primary aim to support the implementation of direct manipulation graphical user interfaces. In this paper, considerable simplifications have been obtained for OSD interfaces with no free moving cursor.

[15] reports a similar tabular specification of interactors with a direct correspondence to a formal interactor model [14]. That interactor model and the specification scheme were not appropriate in our context as they assumed event based communication and do not help ensure the separability of behaviour and presentation components.

The component interface specifications recommended in this paper consist only of syntax. The robustness, clarity and correctness of interface specifications can be enhanced by some increased formality, e.g. by enriching interface descriptions with pre- and post- conditions [19], or even by specifying the temporal sequencing of method invocations as in [15]. This possibility, while not ruled out for future work, conflicts with current industry wide standards such as COM [16].

## 6 Conclusions

This research has focused on the definition of components and their interfaces. Not sufficient emphasis has been put on higher order architectural issues, such as the protocols applied for the communication between components, patterns of their composition and the roles components play in such composition patterns. These questions are the topic of current work which aims to provide notational and tool support for the composition of components.

Our experience is that design concepts for basic interaction tasks are very succinctly described using the interaction styles template [9]. To move from design to development, this design oriented description needs to be matched with a specification of the software components that implement the design concept. This paper has outlined a software architecture for the specification and implementation of graphical interaction design concepts. This architecture, which is a refinement and a 'componentisation' of the PAC model, distinguishes components supporting the 'look' and the 'feel' for a particular design concept.

This paper has reported part of our work in the area of componentising user interface software. Much of the difficulty arose because of the special requirements of the application domain and the limitations of the target platforms, e.g., memory limitations and the requirement for separable behaviour and presentation components. Further, the paper has discussed how the trend towards componentisation of software affects the application standard object oriented architectures such as PAC.

## 7 References

- [1] Jansen, A. (1998). User interface management systems for consumer electronics products: The G+4 Approach. 4<sup>th</sup> Philips Software Conference, June 1998, Eindhoven, The Netherlands, Philips Internal Report.
- [2] Abowd, G. (1992). *Formal aspects of human computer interaction*. Ph.D.Thesis, University of Oxford.
- [3] Bass, L & Coutaz, J. (1991). *Developing software for the user interface*. Addison Wesley.
- [4] Bass, L., (1992). A Metamodel for the run time architecture of an interactive system, *SIGCHI Bulletin*, 24(1).
- [5] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal M. (1996) *A system of patterns. Pattern-oriented software architecture*. Wiley.
- [6] Coutaz, J., (1987). PAC, an object oriented model for dialog design. In Bullinger, H.J. & Shackel, Eds., *INTERACT'87*, North Holland, Elsevier, 431-436.
- [7] Coutaz, J., (1997). PAC-ing the user interface architecture. In M.D.Harrison and J.-C.Torres, Eds., *Design Specification and Verification of Interactive Systems '97*, Springer, 13-28.

- [8] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley.
- [9] Hamberg, R, ter Horst, H., de Ruyter, B., & de Vet, J., (1998) Menu Interaction Styles, An information model and editor, *Nat.Lab. Technical Note 042/98*, Philips Internal Report.
- [10] Hartson, R., Siochi, A.C., Hix, D., (1990). The UAN: A user - oriented representation for direct manipulation systems, *ACM Transactions on Information Systems*, 8, 181-203.
- [11] Hussey, A. & Carrington, D (1996). Using Object-Z to compare the MVC and PAC architectures. In Roast, C & Siddiqi, J (Eds.) *Formal Aspects of the Human Computer Interface*, BCS-FACS workshop, Springer, eWiC series.
- [12] ISO(1989). Information processing systems-open systems interconnection. – LOTOS A formal description technique based on the temporal ordering of observational behaviour. *ISO/IEC 8807*, International organisation for Standardisation, Geneva.
- [13] Krasner, G.E. & Pope, S.T. (1988). A cookbook for using the Model-view-controller User Interface Paradigm in the Smalltalk-80 System. *Journal of Object-Oriented Programming*, 1(3), 26-49.
- [14] Markopoulos, P. (1998). Formal architectural abstractions for interactive software. *Int. Journal of Human Computer Studies*, 49, 675-715.
- [15] Markopoulos, P., Papatzani, G., Johnson, P. and Rowson, J. (1998). Validating semi-formal specifications of interactors as design representations. In Markopoulos, P. and Johnson, P. (Eds.) *Design, Specification and Verification of Interactive Systems '98*, Springer, 102-133.
- [16] Microsoft COM homepage. <http://www.microsoft.com/com>
- [17] Ommering Van, R., (1998). Koala: a Component Model for Consumer Electronics Product Software. In Van der Linden, F., (Ed.). *Development and Evolution of Software Architectures for Product Families*. Springer, LNCS 1429, 76-86.
- [18] Shaw, M. and Garlan, D. (1996). *Software Architecture. Perspectives on an emerging discipline*. Prentice Hall (New Jersey).
- [19] Szyperski, C., (1997). *Component Software. Beyond Object-Oriented Programming*. Addison-Wesley.
- [20] Tarpin-Bernard, F., David, B.T. (1997) AMF: a new design pattern for complex interactive software? In Smith, M.J., Salvendy, G., and Koubek R.J. (Eds.) *Design of Computing Systems: social and ergonomic considerations. Proc. HCI International'97*, Elsevier, 351-354.