An Approach Integrating two Complementary Model-based Environments for the Construction of Multimodal Interactive Applications

David Navarre, Philippe Palanque, Pierre Dragicevic & Rémi Bastide
LIHS-IRIT, Université Paul Sabatier, F-31062 Toulouse Cedex, France
{dragice, navarre, palanque, bastide}@irit.fr
http://lihs.irit.fr/{dragice, palanque, navarre, bastide}

Abstract. This paper presents a tool suite for the engineering of multimodal Post-WIMP Interactive Systems. The work presented here extends previous work done on design, prototyping, specification and verification of interactive systems and integrates two previously unrelated approaches. The first element of this integration is ICoM (a data-flow model dedicated to low-level input modelling) and its environment ICON which allows for editing and simulating ICoM models. The other element is ICOs (a formal description technique mainly dedicated to dialogue modelling) and its environment PetShop which allows for editing, simulating and verifying ICOs models. This paper shows how these two approaches have been integrated and that this integration allows for engineering multimodal interactive systems. We show on a Range Slider case study how these tools can be used for prototyping interactive systems in general and multimodal interaction techniques in particular. We also present in details how the changes in the interaction techniques impact the models at various levels of the software architecture.

Keywords. Interactive Systems Engineering, Multimodal interaction, Prototyping, CASE tools Formal methods.

1 Introduction

According to the recurring desire of increasing the bandwidth between the interactive system and the users more sophisticated interaction techniques called Post-WIMP have been proposed. Such proposals are usually presented in conferences such as ACM CHI (Human Factors in Computing Systems) or UIST (User Interfaces Software and Technology) with a focus on the invention and on the usability evaluation of interactive systems proposing such interaction techniques. Once published, it remains a long way for these interaction techniques to reach the maturity level required for dissemination in industry. One way to deal with this issue is to follow the same path as in the field of software engineering where modelling activities and model-based approaches take the lead with standards like UML. In the field of human computer interaction model-based approaches have been heavily criticized. For instance, at a panel at UIST 94 [52] Dan Olsen wrote “There have been two major problems with this approach [model based ones]. The first is that we always seem to be modelling the interface styles of the previous generation of user interfaces. The second is that the models developed in some way limit the kind of user interface that is possible.” Recent contributions in the field of model-based approaches have been explicitly addressing this issue of coping with new interaction technique. The aim of the work presented in this paper is to present an approach that is able to support prototyping activities for post-WIMP [57, 12] interaction techniques and to integrate them within interactive systems development. To this end we have integrated work done on low-level input handling [22] together with work on formal description techniques [11, 44] of other components of an interactive application. This paper mainly addresses the issue of feasibility meaning that it focuses on what is needed for being able to address the model-based description of highly interactive applications. The issue of how the models can be constructed and what are the skills required are also very relevant but goes beyond the scope of the paper. We can however report the actual use of the tools and the notations by software engineers in two industrial environments that have been reported in [42] and [38].

The integrated approach presented in this paper solves problems that have been identified in each work that has been carried out separately. The work from ICON on low-level input handling, addresses in a very efficient way (via data-flow modelling), the management of flow of events produced by input devices. However, this data-flow paradigm makes it difficult to deal with the representation of the set of states the system can be in and how events produced by the user through the use of input devices make the states evolves. This is precisely what is particularly well represented by the ICO formal description technique. This aspect of low-level handling can however be easily managed with ICOs when only simple interaction techniques are considered. In order to manage interaction techniques that were needed for the industrial projects we have been working on, we had to address both formal modelling of the applications and efficient handling of input devices. The need for integrating data flows (also called continuous events) and state-based representations has been already proposed in [33] for example. However, the integration was made using an automaton notation making it possible, for instance, to represent concurrent interaction techniques (such as two handed interaction).

1 This paper is an extended version of the final version of a paper accepted at EHCI-DSVIS 2004, Lecture Notes in Computer Science n° 3425.
The paper is structured as follows. Section 2 is a brief introduction to model-based approaches for interactive systems engineering. Section 3 presents the Input Configuration approach that is dedicated to low-level input handling in post-WIMP interactive systems. Section 4 recalls the Interactive Cooperative Objects formalism and its environment PetShop. In these sections, the two model-based approaches are exemplified on the same simple case study of the rubber banding. Section 5 details a generic framework for the integration of these two approaches. Section 6 introduces the range slider case study that is used to show that the model-based approaches that we propose can deal with non-trivial interface components and innovative interaction techniques. Section 7 shows how to modify that case study to allow for multimodal (two-handed) interaction. For space reasons, only such a multimodal interaction technique is presented here while several others (including voice and gesture) have been dealt with in a similar way.

2 Model-Based Approaches

Model-based approaches for interactive systems engineering promote the use of models for the design, specification and validation (sometimes through verification) of interactive systems. Figures in this section position related work in the field according to three criteria:

- the evolution (according to time) of the underlying formal description technique used for describing interactive application,
- the part of the interactive application that has been explicitly addressed by the notation. These parts correspond to elements of architectural models like Seeheim [48] and Arch [4]. Sometimes a notation would have been able to address a wider part of the interactive application but we only represent here what has been made explicit by the authors in the referenced paper,
- the type of interaction technique that has been dealt with in the referenced paper.

Since the late sixties a lot of work has been devoted to the definition of notations and tools to support interactive systems design at a higher level of abstraction than implementation code. Figure 1 shows some approaches that have been proposed for the modelling of interactive systems over the years. As the complexity of interactive application was growing, the expressive power of the underlying formalisms has also been increased. Indeed early work from Parnas [47] based on finite state automaton was only able to model application featuring a small number of states. In order to deal with large scale applications this restriction has been removed by the use of more expressive notations like Petri nets [58, 5, 53]. Over the years, other notations based on finite state automaton have been proposed such as [18] that uses statecharts for describing the behaviour of interactive components but as stated in [14] “… some investigation into extensions of finite state machines to enable them to provide this power led to the realisation that Petri nets already do this”.

![Figure 1. Model-based approaches and notations for Interactive Systems engineering.](image)

![Figure 2. Model-based approaches related to architectural models.](image)
According to architectural models (such as Seeheim model [48] or Arch Model [4]) that promote the splitting of interactive systems in several components, model-based approaches for interactive systems have been addressing some (or all) of these components. Indeed, previous work in this field can be classified according to these architectural models as shown in Figure 2. Early work in this field has only focussed on one specific part, and the dialogue part is that which has received much more attention. This is because previous work in the field of computer science can be more directly reused. This Figure also represents the fact that some approaches are broader than other ones addressing several layers of interactive systems. Figure 2 shows a classification of the approaches presented in Figure 1 according to the right-hand side components of the Arch model (i.e. only Dialogue, Logical interaction and Physical interaction).

Figure 3 shows a classification of the approaches presented in Figure 1 according to different kinds of interaction techniques. The Y-Axis on the 3 figures is not conveying additional information. The spatial disposition has been only used to represent the information that would have been otherwise overlapping.

3 Input-Configurations Modelling and Prototyping

ICON (Input Configurator) is a tool for designing input-adaptable interactive applications, i.e., applications that can be controlled with a wide variety of alternative input devices and techniques. ICON provides an interactive editor for the ICOM (Input Configuration Model) graphical notation. In this section, we give a brief overview of the ICOM notation and the ICON visual prototyping environment. More details on the notation and its associated tools can be found in [22, 23, 24].

3.1 Overview of the ICOM notation

The ICOM (Input Configuration Model) notation describes low-level input handling using interconnected modules, with reactive data-flow semantics. In this section, we briefly describe the main features and concepts behind ICOM.

Input Configurations

Devices and slots. ICOM’s main building blocks are devices, which are a broad generalization of input devices: ICOM devices can produce output values, but can also receive input values. Figure 4 shows on the left the graphical representation of a device. A device has typed channels called input slots and output slots, each type having a distinct graphical representation (e.g., circle for Booleans, triangle for integers). Slots can be hierarchically grouped to form structured types, as shown on Figure 4. Black icons on the right-hand side of the element named “Device” represent such slots. The hierarchy is represented by the small vertical lines meaning that the slots underneath correspond to a hierarchical decomposition of the upper one.

Implicit I/O. Whereas the basic behaviour of an ICOM device is processing input values into output values, alternative behaviour is shown on the device by the presence of “notches” (see upper left-hand side of Figure 4 labeled implicit input and implicit output). Non-deterministic devices are described as having implicit input, i.e., additional source of information not fully described by its set of input slots. Example of such devices include devices which are producing data on their own (physical input devices), or asynchronous devices which are temporally non-deterministic. Similarly, devices having implicit output produce alternative effects in addition to simply putting values on the output slots. Examples are devices that manipulate application objects, or devices producing graphical or sound feedback. Indeed, such elements do not appear in the ICOM model and cannot be connected to the rest of the model.
Connections. An input slot of a device can be linked to one or several compatible output slots of other devices by connections, which are represented by wires (see Figure 4). ICON’s execution model forbids multiple connections on the same input slot, as well as connections that generate cyclic dependencies. This is meant to prevent indeterminism in the models.

**Device**

**Input Configuration**

![Diagram showing connections between devices](image)

**Figure 4.** Elements of the ICoM notation. Input ports are displayed in white, output port are displayed in black.

Types of devices. There are three main categories of devices: System devices describe system resources such as input peripherals; Library devices are system-independent utility devices such as processing devices and adapters; Application devices are devices that control a specific application. Toolkit devices are related to the underlying toolkit of the development platform. Each of these devices is presented with more details in section 3.2.

Input configurations. An input configuration is defined by a set of system and application devices, as well as a set of library devices and connections which map the system devices to the application devices (the right-hand side of Figure 4 represent an input configuration including 3 system devices, 7 library devices and 5 application devices).

ICON is modular, and subparts of an input configuration can be encapsulated into compound devices. For example, an input device and a feedback device can be connected then grouped to form a compound device having both external input and external output.

ICoM’s Execution Model

Whereas the contract of a device is to update its output slots every time it is asked to, ICoM’s execution model describes which devices must be triggered and when, and how values are propagated to other devices. The propagation mechanism used, described in [24], is very simple and effective for handling the flow of information.

ICoM’s execution model follows the semantics of reactive synchronous languages such as Esterel [13] or Lustre [29], in which information propagation is conceptually instantaneous. In reactive systems, the environment (e.g., the source of input signals) is the master of the interaction, as opposed to conversational systems in which clients wait to be served.

Describing Interaction Techniques as Input Configurations

From ICoM’s point of view, interaction techniques are transformation flows with feedback. Figure 5 shows the interpretation of keyboard actions in a text editing context (slots are not represented). A “key press” action is transformed into a raw key code through device A, then into a localized symbol through device B. Device C generates strings from combinations of symbols. Those strings are finally used by the text editor. Besides encapsulating a processing function, each device provides its own abstraction of the keyboard. For example, a text component expects to receive strings but the insertion point is best controlled with a lower-level keyboard. ICoM models feedback using
implicit input and output. Figure 6 gives an example of scrolling through a document, and shows the feedback loop through implicit I/O. The Mouse device receives implicit input from the user, the Cursor device produces immediate feedback towards this user, and the Scrollbar tells the application to update its document view.

![Diagram](image)

Figure 5. Handling keyboard input and providing two different feedbacks: the text and the cursor for the insertion point.

![Diagram](image)

Figure 6. Double feedback flow while scrolling through a document (feedback on mouse cursor and feedback on the document (via the scrollbar)).

### 3.2 The ICON environment

The ICON (Input Configurator) Input Toolkit contains an extensible set of system devices and library devices for building input configurations. Devices in ICON are software components that are integrated in the environment. It provides a reactive machine for executing them, as well as a graphical editor for rapid prototyping. ICON is written in Java, and uses native libraries for managing input devices. In this section, we briefly describe the main features of ICON.

**ICON Devices**

*System devices.* ICON’s system devices provide a low-level view of standard and alternative input devices. Under Microsoft Windows operating systems, ICON currently supports multiple mice, graphical tablets, gaming devices and 3D isometric controllers, speech and gesture recognition, and MIDI controllers. System output devices are also available, such as Midi devices for playing music on soundcards, or speech synthesis devices.

*Library devices.* The ICON toolkit has a set of built-in utility devices including mathematical and boolean operators, signal processing devices, type and domain adapters, and devices for conditional control and dispatch. It also provides a set of graphical feedback devices such as cursors and other components, which support for instance, overlay animation on top of Swing frames.

*Toolkit devices.* ICON provides a set of “Swing devices” for controlling existing Java applications that have no knowledge of ICON. One device allows generic control of any Swing widget by sending them mouse and keyboard events, whereas a set of widget-specific devices allow moving scrollbars programmatically or sending strings and caret commands to text components. Event dispatching strategies such as picking and focus are also encapsulated into individual devices.

*Application devices.* Developers can enhance controllability of their application by implementing devices that are specific to their application. Writing an application device is quite straightforward, and mainly requires declaring a set of input slots and implementing an “update” method which is automatically called each time an input slot has received a signal [24]. The production of the “update” method is not as easy. Indeed, it has to be implemented using a programming language (Java for example). The rest of the description only provides the set of input information and events that trigger the method call. This is a known weak point of ICON i.e. it handles very efficiently input devices, event flow and data-flow but a lot of the behavior of the application is left to the actual code of these update methods. This is a reason why, in this paper, we propose to connect ICON to the ICO formal description technique.
The Input Editor
ICON configurations can be built or modified by direct manipulation through a graphical editor. An early prototype of this editor has been described in [22]. In this contribution, the authors showed how the behavior of a standard mouse/keyboard configuration could be easily changed using the editor and its dedicated interaction techniques. In [24], we also gave a subset of interaction techniques that can be described with our graphical notation and directly built using ICON.

The Figure 7 shows a screenshot of the Input Editor window. Library devices and available system and application devices are listed on the left pane, and organized in folders just like a file system. Clicking on a folder (top left pane) displays the devices it contains (bottom left pane). Those devices are dragged on the editing pane to be used. The minimalist input configuration shown on the editing pane of the Figure 7 describes how a freehand tool from a drawing application called ICONDraw [22] is controlled using the mouse. The "sum" devices convert relative (delta) positional values sent by the low-level mouse into absolute values.

The toolbar on the top of the window contains two buttons for executing and stopping the configuration. Execution is fast and does not need compilation, thus allowing easy testing and refinement of input configurations.

One simple example: One-Handed and Two-Handed Rubber Banding
ICON’s graphical editor allows the application designer to quickly build and test input configurations that make use of alternative sets of physical input devices, or modify existing configurations to adapt to enriched or impoverished input. The Figure 8 illustrates how a conventional technique can be changed into a Post-WIMP technique when a new input device (a graphical tablet) becomes available. The left upper part of the Figure 8 shows the part of ICONDraw’s default input configuration which describes the standard rubber-banding technique for drawing lines: the user indicates the first end of the segment by pressing the mouse button, then the other end by dragging and releasing the button. The "firstThen" device encapsulates the simple automaton which implements this behavior. This means that when the mouse is pressed the coordinates x and y of this mouse press are assigned to the first element named “p0”. The other x and y values when either dragging or releasing the mouse button are assigned to the second element called “p”. The arrow between the two configurations show how the configuration has to be changed if a specific device is assigned to a each extremity of the line. As shown on the lower part of the Figure 8, this configuration has then been simplified so that each end of a segment being created is controlled by a separate pointing device. By doing this, the designer has just described a very basic bimanual interaction technique where the WACOM tablet controls the point representing the end of the line (p) and the mouse the first point of the line (p0) (Figure 8 on the right).
4 Dialogue Modelling and Prototyping

This section recalls the main features of the ICO formalism, which we use to model the case study. We encourage the interested reader should look at [5, 11] for a complete presentation of the formal description technique.

4.1 Overview of the ICO formalism

The Interactive Cooperative Objects (ICOs) formalism is a formal description technique dedicated to the specification of interactive systems [11]. It uses concepts borrowed from the object-oriented approach to describe the structural or static aspects of systems, and uses high-level Petri nets [29] to describe their dynamic or behavioural aspects.

Petri Nets is a graphical formalism made up of four components: the state variables (called place, depicted as ellipses), states changing operators (called transitions, depicted as rectangles), arcs, and tokens. Tokens are hold by places; arcs link transitions to places and places to transitions. The current state of a system is fully defined by the marking of the net (i.e., both the distribution and the value of the tokens in the places). For a state change to occur a transition must be fired. A transition is fireable if and only if each of its input places holds at least one token. When the transition is fired, one token is removed from each input place and a token is deposited in each output place.

ICOs are dedicated to the modelling and the implementation of event-driven interfaces, using several communicating objects to model the system, where both behaviour of objects and communication protocol between objects are described by Petri nets. The formalism made up with both the description technique for the communicating objects and the communication protocol is called the Cooperative Objects formalism (CO and its extension to CORBA COCE [10]).

In the ICO formalism, an object is an entity featuring four components:

Cooperative Object (CO): a cooperative object models the behaviour of an ICO. It states how the object reacts to external stimuli according to its inner state. This behaviour, called the Object Control Structure (ObCS) is described by means of high-level Petri net. A CO offers two kinds of services to its environment. The first one, described with CORBA-IDL [55] (Interface Description Language), concerns the services (in the programming language terminology) offered to other objects in the environment. The second one, called user services, provides a description of the elementary actions offered to a user, but for which availability depends on the internal state of the cooperative object (this state is represented by the distribution and the value of the tokens (called marking) in the places of the ObCS).

Presentation part: the Presentation of an object states its external appearance. This Presentation is a structured set of widgets organized in a set of windows. Each widget may be a way to interact with the interactive system (user ⇒ system interaction) and/or a way to display information from this interactive system (system ⇒ user interaction).

Activation function: the user ⇒ system interaction (inputs) only takes place through widgets. Each user action on a widget may trigger one of the ICO's user services. The relation between user services and widgets is fully stated by the activation function that associates to each couple (widget, user action) the user service to be triggered.
**Rendering function:** the system → user interaction (outputs) aims at presenting to the user the state changes that occurs in the system. The rendering function maintains the consistency between the internal state of the system and its external appearance by reflecting system states changes.

ICO are used to provide a formal description of the dynamic behaviour of an interactive application. An ICO specification fully describes the potential interactions that users may have with the application. The specification encompasses both the "input" aspects of the interaction (i.e., how user actions impact on the inner state of the application, and which actions are enabled at any given time) and its "output" aspects (i.e., when and how the application displays information relevant to the user). Time-out transitions are special transitions that do not belong to the categories above. They are associated with a timer that automatically triggers the transition when a dedicated amount of time has elapsed. When included in a system model such transition is considered as a system transition. They can also be included in a user model representing spontaneous user's activity.

An ICO specification is fully executable, which gives the possibility to prototype and test an application before it is fully implemented [38]. The specification can also be validated using analysis and proof tools developed within the Petri nets community and extended in order to take into account the specificities of the Petri net dialect used in the ICO formal description technique.

### 4.2 ICO Models for a rubber banding drawing application

The **rubber banding** is a very classical interaction technique used in most graphical tools. It allows a user to draw a line (or a shape) based on the "drag and drop" interaction technique, where, while dragging, a temporary line is drawn, called ghost. We present here, through this classical example, the four parts of an ICO specification: the behaviour, the presentation part and the link between them stated by the activation and the rendering function.

1. **Behaviour (ObCS).** The behaviour of the rubber banding application is represented by its ObCS shown in Figure 9.
   Initially, the application is in an idle state (a token is set in place `Idle`). When the mouse button is pressed (transition `BeginDrag`), it starts the drawing of a ghost that is updated while moving the mouse pointer (token in place `Dragging`) by the firing of the transition `Drag`. When the mouse button is released (transition `EndDrag` fired), the ghost is erased, the definitive line is drawn, and the application returns in its idle state.

![Figure 9 - Behaviour of the rubber banding drawing application.](image)

2. **Presentation part.** The presentation part described the external presentation part of the Range Slider. We describe hereafter (Figure 10) a set of basic rendering methods that characterise the DrawablePanel. This set of methods is used to produce the rendering by the rendering function (see the point 3).

```java
Class DrawableJPanel
  Rendering methods {
    drawGhost(int x0, int y0, int x1, int y1) {
      //Draw a dashed line between point (x0, y0)
      //and point (x1, y1).
    }
    eraseGhost(int x0, int y0, int x1, int y1) {
      //Erase the dashed line drawn between
      // point (x0, y0) and point (x1, y1).
    }
    drawLine(int x0, int y0, int x1, int y1) {
      //Draw a line between point (x0, y0)
      //and point (x1, y1).
    }
  }
```
3. Rendering Function. The rendering function describes how state changes impact the presentation part of the application. As state changes are linked to token movements, rendering items may be linked to either place or transition. Figure 12 describes the rendering function for the rubber banding application. The first line, for instance, shows that when a token enters the place Dragging, the corresponding rendering is to draw a ghost between the coordinates brought by the token.

<table>
<thead>
<tr>
<th>ObCS element</th>
<th>Feature</th>
<th>Rendering method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place Dragging</td>
<td>Token (&lt;x_0, y_0, x_1, y_1&gt;) Entered</td>
<td>drawGhost((x_0, y_0, x_1, y_1))</td>
</tr>
<tr>
<td></td>
<td>Token (&lt;x_0, y_0, x_1, y_1&gt;) Removed</td>
<td>eraseGhost((x_0, y_0, x_1, y_1))</td>
</tr>
<tr>
<td>Transition EndDrag</td>
<td>Fired with (&lt;x_0, y_0, x_1, y_1&gt;)</td>
<td>drawLine((x_0, y_0, x_1, y_1))</td>
</tr>
</tbody>
</table>

Figure 11 - Rendering function of the rubber banding application.

4. Activation Function. The activation function (shown by Figure 12) relates the events produced by a widget to the transitions of the ObCS. Thus if the transition is fireable and the event is produced (by a corresponding user action on the widget) then the transition is fired (and its action is executed).

<table>
<thead>
<tr>
<th>Widget</th>
<th>Event</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Panel</td>
<td>Move</td>
<td>Move</td>
</tr>
<tr>
<td>Panel</td>
<td>MouseDown (&lt;x, y&gt;)</td>
<td>BeginDrag</td>
</tr>
<tr>
<td>Panel</td>
<td>MouseDrag (&lt;x, y&gt;)</td>
<td>Drag</td>
</tr>
<tr>
<td>Panel</td>
<td>MouseReleased (&lt;x, y&gt;)</td>
<td>EndDrag</td>
</tr>
</tbody>
</table>

Figure 12 - Activation function of the rubber banding application.

4.3 Overview of PetShop Environment

In this section we present how the PetShop environment supports the editing of the various elements presented in the previous paragraph. Some screen shots are included in order to show the current state of the tool with respect to the overall architecture presented in Figure 13. Plain arrows represent the information exchange between a tool and a file (for instance the ObCS graphical editor tool reads and writes the ObCS). The dashed lines represent information flows that can be done in parallel (for instance the exploitation of the ObCS is done in parallel by the analysis tool and the ICO interpreter).

ObCS Editor
Our approach is supported by a tool call PetShop which includes a distributed implementation of high-level Petri net interpreter written in Java. All the components of the ObCS can be directly built using PetShop. PetShop also automatically generates an Object Petri net from the IDL (Interface Description Language) description [29]. The editing of the Object Petri net is done graphically using the Palette in the center of the toolbar. The left part of the toolbar is used for generic functions such as load, save, cut copy and paste. The right hand side of the toolbar drives the execution of the specification (see Figure 32).

Edition of the Presentation
Currently, PetShop is linked to the JBuilder environment for the creation of the presentation part of the ICOs. Thus creation of widgets is done by means of the JBuilder interface builder. However, we have not yet created a visual tool for editing the rendering and the activation function that still have to be programmed in Java.
Execution environment
A well-known advantage of Petri nets is their executability. This is highly beneficial to our approach, since as soon as a behavioural specification is provided in term of ObCS, this specification can be executed to provide additional insights on the possible evolutions of the system.

Figure 32 shows the execution of the specification of the Range Slider in Petshop. The ICO specification is embedded at run time according to the interpreted execution of the ICO. At run time user can both look at the specification and the actual application. They are in two different windows overlapping in Figure 32 The window RangeSlider corresponds to the execution of the window with the Object Petri net underneath.

In this window we can see the set of transitions that are currently fireable (represented in dark grey and the other ones in light grey). This is automatically calculated from the current marking of the Object Petri net. Each time the user acts on the RangeSlider the event is passed on to the interpreter. If the corresponding transition is fireable then the interpreter fires it, performs its action (if any), changes the marking of the input and output places and performs the rendering associated (if any).

5 Coupling Input Configurations and Dialogue
In the introduction we have presented the respective advantages and limitations of each notation as well as the rationale for integrating them. This section presents how the two approaches have been effectively integrated. We show first how this coupling takes place at the model level (ICOM and ICOs) and then at the environment level (ICON and PetShop).

5.1 Models Coupling: ICoM and ICOs
Whereas ICO’s activation functions list the couples Widget × Event and the user services they trigger, ICoM describes how each event is produced. We depict here the two levels of coupling between ICoM and ICO by describing both activation devices and service devices.

Activation devices
ICoM primarily interfaces with ICO at the activation function level in the following way:

- Each widget in the activation function has a device counterpart in ICoM we call activation device,
• For a given widget, each event name is an input slot in the corresponding device,
• A non-valued event is denoted by a Boolean slot on the device,
• A event of type \(<x_1, \ldots, x_n>\) is denoted by \(n\) slots with the corresponding types on the device.

![Figure 14. Activation function of the rubber banding and its associated activation device.]

An example of activation function and the matching activation device is given on the Figure 14. Using this “interface”, the low-level part of the rubber banding technique can be described by connecting ICO/M devices. As ICO events are totally separated from the AWT toolkit events, there is not necessarily a relationship between e.g., the ICO MouseDown in our example and the AWT MousePressed. For example, the MouseDown may be issued by a graphical tablet, a joystick, or even a voice command. In fact, ICO events are of higher level of abstraction compared to mouse and keyboard events. The same is true for ICO widgets which do not necessarily have to be associated with actual instances of AWT widgets, and may be simply used to group abstract events.

**Service devices**

As explained before, the interface provided by activation devices allows to describe how ICO events are produced using the ICO/M notation. However, this interface alone is not sufficient for describing all input mechanisms: for example, most WIMP interfaces use a picking mechanism to determine the widget the mouse operates on. In our model, we would suppose that picking is a service offered by ICO to ICO/M, and that this service is encapsulated into a device. We will give other examples of service devices in our case study of range slider.

There are two important differences between activation devices and service devices: first, activation devices are pure output devices whereas service devices can be input, output or input/output devices. Second, activation devices can be automatically deduced from the ICO specification whereas query devices must be described separately. This is partly due to the fact that the services needed are highly dependent from what is being described.

**5.2 Systems Coupling: ICON and PetShop**

To be visible to ICON, a Petshop application must implement the WidgetSet interface that exposes the domain of definition of its activation function. This interface has one unique method returning all widgets the application uses, which is sufficient to retrieve all possible Widget × Event couples as each widget exposes the set of events it is able to produce. On the ICON side, a class of device folder has the responsibility to create activation devices given a WidgetSet passed in its constructor. Thus, the great part of the process of generating the interface between ICON and Petshop is automated. Only service devices must be written by hand.

**5.3 Execution Coupling**

In addition to implementing the WidgetSet interface, ICON/Petshop applications use the same code which creates activation devices and starts a default input configuration. These applications are launched from the PetShop environment.

While running, an input configuration can be deactivated. This is essential as ICON allows redefining input handling at a very low-level, which can possibly hang all the system. For similar reasons, input configurations can be edited while paused but not while running. In contrast, the editing of the ICO part with Petshop is fully dynamic.
6 Case Study of a conventional RangeSlider

In order to present the tool suite we have developed for the engineering and very-high prototyping of multimodal interactive systems, we use later a case study. We first present the case study offering standard interaction technique (in this section) and show how to this case study can be easily extended in order to be manipulated through different multimodal interaction techniques.

Figure 15. Screen shot of the Film application, showing four Range Sliders.

Figure 15 presents the window of the film application, generated using the SpotFire environment (http://www.spotfire.com). This window is split into a zoomable starfield display on the left and a set of query devices on the right [2]. Query devices allow dynamic control of information shown on the starfield display. The two first query devices on the picture are Range Sliders. A Range Slider is an interactive component that allows easily selecting a range of values (RV) within a range of possible values (RPV). Figure 16 shows in detail the presentation part of this interactive component.

Figure 16. The various elements of a Range Slider. This one allows selecting a range of films by their length (the current selection corresponds to all the movies between 68 and 360 mn).

The presentation part is made up of 5 interactive components:

- A central part called lift, that allows the user to set both ends of the RV at a time,
- Two buttons called LeftArrow and RightArrow that allow the user to set precisely the corresponding end of the RV,
- Two buttons called LeftBar and RightBar that allow the user to set rapidly and roughly one of the values at the end of the RV.

The set of possible user actions on these interactive components is:

- Down on interactive component LeftBar at X,Y
- Down on interactive component RightBar at X,Y
Drag on interactive component LeftArrow at X,Y
Drag on interactive component RightArrow at X,Y
Drag on interactive component Lift at X,Y

The RangeSlider class is composed of elements from three other classes. More precisely it is made up of 5 instances: 2 RangeArrows, 1 RangeLift and 2 RangeBars. These three classes inherit from the same abstract class: Button. According to the software engineering principles for interactive systems construction, it is important to separate abstraction from presentation. For this reason a class Range representing the abstract data type of a range has been added to the set of classes.

6.1 Specification of the Range model

Later (Figure 18), we present the class Range as an ICO class. The Range class is the behavioural specification of the IRange interface given in Figure 17.

```java
interface IRange {
    float getMinimum();
    void setMinimum(float value);
    float getMaximum();
    void setMaximum(float value);
    void getAbsoluteBounds(out float absMin, out float absMax);  
}
```

Figure 17. The IRange interface.

The IRange interface is provided with operations to get and set the minimum and maximum values of the range. The ICO specification refines this interface by describing precisely the behaviour for the operations, in the form of a high-level Petri net. All the get operations are related to places by test arcs. These arcs play the role of validating conditions for a transition as the tokens are not removed by the firing of the transition but they are required for its fireability.

This net ensures an invariant on the consistency of the minimum and maximum values of the range, i.e.,

\[ \text{absMin} \leq \text{getMinimum()} \leq \text{getMaximum()} \leq \text{absMax}. \]

6.2 Specification of the RangeSlider

The following sections present the complete specification of the Range Slider. First, we present the set of services of the class then the ObCS and the states. Lastly, the presentation part is described, which includes the widget as well as the activation and rendering functions.
Figure 18. The CO class Range.

Services
For describing the set of services offered by a Class to its environment we use the Interface Description Language (IDL) as defined by the Object Management Group for the CORBA standard Corba [55].

```
interface IRangeSlider {
    }
```

It can be easily seen that the IDL (Interface Description Language) description does not represent the MVC mechanisms such as notification as this is closer to implementation with respect to the specification phase. As the RangeSlider is not used by other classes its interface is empty.

Behaviour (ObCS)
The behaviour of the Class Range Slider is fully described by its ObCS. The Figure 19 on the next page describes the set of places (and their type) and the set of transition (and their action) of the Object Petri net defining the behaviour of the class. In that model the CurrentRangeValue holds a token corresponding to the current value of the range. This token is also responsible for the mutual exclusion of interactive components on the rangeslider interface. Indeed, in this
idle state all the interactive components are available (all the transitions are fireable in the model). If, for example, BeginUpdateLeftValue is fired (corresponding to a user action (mouse pressed) on the left arrow then the only available transition (and thus user interaction) will be moving the mouse (firing the transition UpdateLeftValue). All the other transition are not fireable anymore meaning that the interaction objects will automatically appear as disabled on the user interface. Section 7 will show that adding concurrency in the model (by duplicating the token in place CurrentRangeValue will make it possible to offer multiple devices interaction to the user).

```plaintext
Class RangeSlider
Specifies IRangeSlider {
    //Definition of Places
    Place CurrentRangeValue <Range> = { 1*<range> };
    Place UpdatingAllValues <Range>;
    Place UpdatingLeftValues <Range>;
    Place UpdatingRightValues <Range>;
    //Definition of Transitions
    Transition BeginUpdateAllValues {}
    Transition UpdateAllValues {
        Action {
            v.setMinimum(v.getMinimum() + x);
            v.setMaximum(v.getMaximum() + x);
        }
    }
    Transition EndUpdateAllValues {}
    Transition BeginUpdateLeftValue {}
    Transition UpdateLeftValue {
        Action {
            v.setMinimum(v.getMinimum() + x);
        }
    }
    Transition EndUpdateLeftValue {}
    Transition BeginUpdateRightValue {}
    Transition UpdateRightValue {
        Action {
            v.setMaximum(v.getMaximum() + x);
        }
    }
    Transition EndUpdateRightValue {}
    Transition DirectUpdateLeftValue {
        Action {
            v.setMinimum(x);
        }
    }
    Transition DirectUpdateRightValue {
        Action {
            v.setMaximum(x);
        }
    }
} behaviour
```
Widgets

The widget part describes the external presentation part of the Range Slider. We describe later (see Figure 20) the set of basic attributes that characterise each instance of each component of the RangeSlider. For each attribute, a method set and a method get must be generated. For instance, the attribute range needs the two following methods:

1. `void setScale(float newScale);`
2. `float getScale();`

These methods must be used by other classes to access and modify the values of the attributes. We use inheritance between classes (here, RangeSlider inherits from JComponent defined by java swing), thus providing through inheritance a set of attributes. The attributes described later are only new attributes or the ones that are overridden.

Another part of this widget part is the description of a set of methods used to produce rendering by the rendering function (see next subsection).

```java
class WidgetRangeSlider extends JComponent {
    Attributes {
        float scale //It is the value represented by one pixel
    }
    Rendering methods {
        ShowLeftValue(float v) {
            //Changes the position of the left arrow,
            //the size of the left bar and the lift
        }
        ShowRightValue(float v) {
            //Changes the position of the right arrow,
            //the size of the right bar and the lift
        }
    }
}
```
```java
ShowLeftArrowWorking() {
    // Shows the left arrow as a pressed button.
}
ShowLeftArrowIdle() {
    // Shows the left arrow as normal.
}
ShowRightArrowWorking() {
    // Shows the right arrow as a pressed button.
}
ShowRightArrowIdle() {
    // Shows the right arrow as normal.
}
```

**Figure 20.** The widget description part of the Range Slider.

![Image](image-url)

**Figure 21.** Editing of the Range Slider using the JBuilder Interface builder.

**Rendering Function**

The rendering function describes how state changes are rendered to the user. In the Range Slider specification, rendering is divided in two parts. The first part (see Figure 22) is related to inner state changes of the Range Slider, while the second one (see Figure 23) is related to the Range class as RangeSlider is a view of Range. Indeed, as Range Slider is a view of range, it needs to render state changes of this object. Variable range used in the method call to show the right and left value of the range refers to the object with which a Range Slider is instantiated.

**Inner Rendering**

<table>
<thead>
<tr>
<th>ObCS element</th>
<th>Feature</th>
<th>Rendering method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place</td>
<td>UpdaterLeftValue</td>
<td>Token Entered ShowLeftArrowWorking()</td>
</tr>
<tr>
<td>Place</td>
<td>UpdaterRightValue</td>
<td>Token Entered ShowRightArrowWorking()</td>
</tr>
</tbody>
</table>

**Figure 22.** The inner rendering function of the Range Slider.

**Rendering as view of range**

<table>
<thead>
<tr>
<th>ObCS element</th>
<th>Feature</th>
<th>Rendering method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place</td>
<td>Minimum</td>
<td>Token Entered ShowLeftArrowWorking(range.getMinimum())</td>
</tr>
<tr>
<td>Place</td>
<td>Maximum</td>
<td>Token Entered ShowRightArrowWorking(range.getMaximum())</td>
</tr>
</tbody>
</table>

**Figure 23.** The rendering function of the Range Slider as view of range.
Activation Function
The activation function relates the events produced by a widget to the transitions of the ObCS. Thus if the transition is fireable and the event is produced (by a user action on the widget) then the transition is fired (and its action executed).

As we can see, all the transitions of the ObCS in Figure 24 are related to user actions which means that only the user can trigger Range Slider functions (the class has no spontaneous behaviour and does not offer services to other classes of the application). This can also be seen from the IDL (Interface Description Language) description as no service is offered to the environment.

<table>
<thead>
<tr>
<th>Widget</th>
<th>Event</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lift</td>
<td>Down</td>
<td>BeginUpdateAllValues</td>
</tr>
<tr>
<td>Lift</td>
<td>Move &lt;x&gt;</td>
<td>UpdateAllValues</td>
</tr>
<tr>
<td>Lift</td>
<td>Up</td>
<td>EndUpdateAllValues</td>
</tr>
<tr>
<td>LeftArrow</td>
<td>Down</td>
<td>BeginUpdateLeftValue</td>
</tr>
<tr>
<td>LeftArrow</td>
<td>Move &lt;x&gt;</td>
<td>UpdateLeftValue</td>
</tr>
<tr>
<td>LeftArrow</td>
<td>Up</td>
<td>EndUpdateLeftValue</td>
</tr>
<tr>
<td>RightArrow</td>
<td>Down</td>
<td>BeginUpdateRightValue</td>
</tr>
<tr>
<td>RightArrow</td>
<td>Move &lt;x&gt;</td>
<td>UpdateRightValue</td>
</tr>
<tr>
<td>RightArrow</td>
<td>Up</td>
<td>EndUpdateRightValue</td>
</tr>
<tr>
<td>LeftBar</td>
<td>Click &lt;x&gt;</td>
<td>DirectUpdateLeftValue</td>
</tr>
<tr>
<td>RightBar</td>
<td>Click &lt;x&gt;</td>
<td>DirectUpdateRightValue</td>
</tr>
</tbody>
</table>

Figure 24. The activation function of the Range Slider.

6.3 Interface between the ICO specification of the range slider and ICoM

Each widget in the activation function of Figure 24 is automatically encapsulated into an activation device in ICOn (see Figure 25). Activation devices are a special kind of application devices dedicated to PetShop. They have been built as a software interface between input configurations and PetShop environment (and more precisely the activation function). The 11 slots of those five activation devices cover the input vocabulary defined by the activation function of the Range Slider. We added service devices to allow geometrical picking and conversion of values from the pixel space to the range space (see Figure 25). The first five service devices handle picking for each Swing widget used in the range slider. For example, when the InLift device receives a true signal on its pick input slot, it emits true on its inside output slot if the last received point p is inside the Lift widget. Otherwise it emits false. Points p are given in screen pixel coordinates. Then, when a false signal is received on the pick input slot, a false signal is also emitted on the inside output slot. The pixToModel service device converts x values given in the screen pixel space into range values. It also converts delta (relative) pixel values into range delta values.

Figure 25. The activation devices (1st row) and the service devices (2nd row) of the range slider.

6.4 Input configuration of the conventional range slider

The input configuration of the range slider describes the way it is manipulated with a mouse. Figure 26 shows this configuration: Mouse moves are integrated then used to animate a mouse cursor on top of the application frame. The
cursor also propagates the state of the left mouse button to the rest of the configuration. Shortcuts, represented by grey vertical lines, are used to display the same cursor device at different places of the configuration. The right half of the configuration has five parts, each describing how events of a given widget are produced. For each of those parts, the output of a picking device and $x$ values sent by the cursor are propagated to a defaultMove or defaultSet device. Those devices encapsulate behaviours that are common to a set of widgets, and are described hereafter.

Figure 26. The input configuration describing standard mouse manipulation of the range slider.

Figure 27. Definition of the defaultMove device.
The *defaultMove* device is used to control lift and arrow widgets. It is a compound device defined by the input configuration shown on Figure 27. The *transition* device is used to convert changes in the Boolean state of the *on* input slot into *Down* and *Up* events. The *delta* and *pixToModel* devices are used to convert *x* values given in screen space into changes in the range value. The *pass* device is used to stop production of *Move* events as long as *on* is *false*. Its behaviour is the following: when it receives a *true* signal on its *pass* slot it forwards the last value received. It then forwards all values as long as a *false* is not received on its *pass* slot.

The *defaultSet* device is used to control the two bars and its configuration is shown on Figure 28. The *pixToModel* device is used to convert *x* values in screen space into range values. When *on* becomes *true*, the current range value is sent to the output. Here, a *hasSignal* device is used instead of the *transition* device previously described in order to force emission of a *false* signal after the click has been produced. Here is the explanation: At the tick *t*, *on* receives a *true* signal which is transmitted to *pass*. At the tick *t+1*, *on* keeps the same state but has no more signal, which causes *hasSignal* to emit a *false* signal propagated to *pass*.

7 A Two-Handed Range Slider

This section presents a modification of the case study in order to allow for two handed interaction on the range slider. As stated in the introduction, it is not the purpose of this paper to discuss the usability of such an interaction technique but to describe how the description techniques deal with highly interactive application. Indeed, this section presents in full details how the description techniques are able to address the modifiability aspects of the models and more precisely the impact of moving from monomodal interaction technique to a multimodal one.

7.1 Adding a Second Input Device

We describe a scenario in which the default input configuration is modified to handle two mice. In this scenario, each mouse moves a separate pointer but both pointers will be used the same way to control range slider. This allows both symmetric bimanual interaction and two-user collaborative interaction with the range slider.

The mouse device used in the previous configuration (Figure 26) is the *system mouse*, which blends output from all physical mice. When launched, ICON’s editor also shows on the left pane all currently connected mice as *individual devices*, including PS/2, serial and USB mice (see Figure 29). The user just has to identify the mice he wants to use and drag them in the edition pane. Note that other pointing devices such as graphical tablets can also be used, or even emulated with devices such as keyboard or voice recognition.
As both pointers share the same behaviour, the configuration described in Figure 26 only has to be duplicated and mouse devices replaced. In order to reduce visual complexity and facilitate the addition of the second mouse, it is a better practice to encapsulate it into a compound device (named `defaultControl`). Lastly, two instances of this compound device are instantiated and connected to two separate USB mice, as shown on Figure 30.

**Scenario 1: Supporting Concurrent Input at Dialogue Level**

The modification done on the input devices requires modification at the dialogue level (i.e., in the ICO model on the range slider). Left-hand side of Figure 31 shows the range slider application in use (the full window is visible in the Figure 32). As an extra mouse has been added using ICON two mouse pointers are available. The user has pressed the left arrow and now the only action available to the user is either to move the arrow or to release the mouse button. This means that both mice cannot be used at the same time. The right-hand side of Figure 31 shows how the ICO model has to be modified in order to fully accommodate the additional mouse. This is done by adding a new token in the central place of the model (`CurrentRangeValue`).
Figure 31. Tokens modification to support concurrent interaction on the Range Slider. An arrow has been added to the left screenshot to show up the widget being manipulated, while a cross shows the widget that does not move.

Figure 32 shows how this modification makes it available for the user to use both mice at a time. In the current state of the application both arrows have been pressed. This can be seen on the model as there is a button both in place UpdatingLeftValue and in place UpdatingRightValue. Currently four actions are available to the user (two for each mouse) releasing the mouse button or moving the arrows. This can be seen on the model as four transitions are fireable.

Figure 32. Two-handed manipulation of the Range Slider.
Scenario 2: Constraining Two-Handed Manipulations

This modification at the dialogue level raises another issue. Indeed, it is now possible to modify both min and max values of the range by using the lift of the range slider. Due to the fact that a second mouse is available it is also possible for the user to use that mouse to act on the other interactive elements of the range slider.

The left-hand side of the Figure 33 shows this kind of abnormal behaviour (one mouse is currently used for moving the lift and the other one is used for moving the left arrow). The actual perceivable behaviour on the user interface is that the last action is taken into account and thus the left arrow move chaotically according to user's actions.

This undesired behaviour can be easily modified on the dialogue model by making unavailable the user's actions on the interactive objects of the range slider if the lift is already used. This is modelled by adding inhibitor arcs between place UpdatingAllValues and the other transitions of the net. The right-hand side of Figure 33 shows the modified model featuring four additional inhibitors arcs. As shown on that figure, while the lift is used (a token is present in place UpdatingAllValues), the other transitions are unavailable.

Figure 33. Constraining manipulation by modifying dialogue. One of the two inhibitor arcs has been thickened on the right screenshot to make it more visible.

8 Other interaction techniques and modalities

In addition to the bimanual manipulation described above, we developed other modalities and interaction techniques that we cannot describe here for space reasons. Those include:

- **Speech interaction**: the range slider is manipulated with voice commands increase left, decrease right or stop left, which “animate” range values. In order to build such a speech interaction IcoM model had to be mended by adding a service device providing absolute minimum and maximum of the range slider, in order to make use of higher-level commands such as “left to minimum” or “full range”.

- **Gestural interaction**: we described a technique in which the range slider is manipulated using the left mouse button whereas the right button is used to draw gestures and to issue the same higher-level commands as above.

  Moreover, other advanced techniques already described and implemented with ICON (see for example [22] and [24]) can be directly connected to the range slider without much effort. Those techniques include:

- **Speech and keyboard pointing**: the pointers used to manipulate the range slider may be operated using keyboard keys or speech commands. Only the low-level part of the standard configuration described in Figure 26 has to be modified to make use of those accessibility configurations.

- **Filtered pointing**: by inserting filters just after the pointing device (Figure 26) a wide range of techniques we call “filtered pointing” have been described. For example “inertia” filters make it possible to set range values to their minimum or maximum using natural “throw” gestures.
9 Conclusion

This paper has presented a tool suite dedicated to the engineering of multimodal interactive systems. Thanks to the ICOs formalism, the approach proposed deals with the functional core and the dialogue part of multimodal interactive systems. Thanks to the ICON notation the approach deals explicitly with input devices and input configurations. As these two models are supported by dedicated editing, simulation and execution environments, we have shown how very high fidelity prototyping can be performed and its related impact at various levels of the Arch architectural model.

As stated in the introduction, the integrated approach presented in this paper has been applied in the field of command and control systems for safety critical applications. We have seen that the use of two different notations and two different environments (even though quite seamlessly integrated) induce a quite high learning curve for the engineers working on the projects. However, the solution offered substantial benefits both in terms of prototypability of interaction techniques and reliability of the resulting systems that would have been otherwise impossible to achieve.

This work belongs to a more ambitious project dedicated to the engineering of multimodal interactive systems for safety critical applications including military aircraft cockpits and satellite ground stations. The aim of this work is not only to provide notations and tools for building multimodal interactive systems but also to support verification and validation in order to support certifications activities that are a critical phase in the development process of interactive safety critical applications.

10 Acknowledgements

The work presented here is partly funded by French defence agency (Direction Générale pour l’Armement) under contract n° 00.70.624.00.470.75.96. The work presented here is partly funded by the French Space Agency CNES (Centre National d'Etudes Spatiales) under the R&T action n°CC201*02. Special thanks are due to Didier Bazalgette for precise information about the field of command and control systems for military applications.

11 References

15. Publisher Academic Press Ltd. London, UK, UK