A Tool-Supported Design Framework for Safety Critical Interactive Systems

Rémi Bastide, David Navarre & Philippe Palanque
LIIHS-IRIT, University Toulouse 3, 118 route de Narbonne, 31062 Toulouse Cedex 4, France. {navarre, palanque, bastide}@irit.fr

Abstract
This paper presents a design framework for safety critical interactive systems, based on a formal description technique called the ICO (Interactive Cooperative Object) formalism. ICO allows for describing, in a formal way, all the components of highly interactive (also called post-WIMP) applications. The framework is supported by a case tool called PetShop allowing for editing, verifying and executing the formal models. The first section describes why such user interfaces are challenging for most description techniques, as well as the state of the art in this field. Section 3 presents a development process dedicated to the framework. Then, we use a case study in order to recall the basic concepts of the ICO formalism and the recent extensions added in order to take into account post-WIMP interfaces’ specificities. Section 5 presents the case tool PetShop and how the case study presented in the previous section has been dealt with. Lastly, we show how PetShop can be used for interactive prototyping.

1 Introduction
Safety critical systems have been for a long time the application domain of choice of formal description techniques (FDT). In such systems, where human life may be at stake, FDT are one of the means of achieving the required level of reliability, the other classical ones being redundancy and testing. The perceived cost of application of FDT is justified by the increased confidence in the good functioning of the system they can provide.
A noticeable trend we are now witnessing is the appearance of highly interactive user interfaces in the field of safety critical systems, such as airplane cockpits, air traffic control workstations, nuclear plant control systems, etc. In turn, the appearance of such advanced user interfaces emphasizes the need to properly account for human factors in these new systems, and to ensure their usability as well as their reliability.

More than ever, highly interactive safety critical systems call for methods allowing both verification and validation: Verification answers the question "Are we building the system right?". Does the system verify crucial properties? With the help of FDT, such questions can be answered through mathematical analysis of the specifications.
Validation answers the question "Are we building the right system?". Does the system meet the requirements? Is it usable and useful? When the emphasis is on validation, iterative design processes [8] are generally put forward with the support of prototyping as a critical tool [17]. However, if used in a non-structured way, and without links to the classical phases of the development process, results produced using such iterative processes are usually weak in terms of reliability. They also can be unacceptable when interfaces for safety critical applications are concerned.

The work described in this paper aims at reconciling the concerns of reliability and usability. To this end we present a framework that consists in a formal specification technique (Interactive Cooperative Objects, ICO), a tool (PetShop) supporting the ICO notation, and a design process using both formal specification and user-centered design, allowing for both verification and
validation. This framework is presented through a case study in the domain of air traffic management, typical of the recent development of highly interactive safety critical systems.

Several approaches propose solutions for the reconciliation of the specification and the validation phases in the field of interactive applications, but these solutions are often incomplete.

**Interaction style viewpoint:** post-WIMP user interfaces are not yet widely developed. For this reason, most of the approaches (see for instance [9]) only deal with WIMP interfaces (Window Icon Menu and Pointing), i.e. static interfaces for which the set and the number of interactors is known beforehand. The behaviour and the role of these interactors is standardised (typically windows and buttons belong to this category).

**Development phase viewpoint:** we often find disparate solutions that do not integrate the various phases in a consistent manner [13]. So, most often several gaps remain to be bridged manually by the teams involved in the development process.

**Reliability of results viewpoint:** several integrated approaches have been proposed for WIMP interactive applications. Among them we find TRIDENT [4] which is the more successful one as it handles both data and dialogue description and as it also incorporates ergonomic evaluation by means of embedded ergonomic rules. However, specification techniques used in the project have not been provided with analysis techniques for verifying models and the consistency between models.

The paper is structured as follows: Section 2 discusses the issues raised by the description of highly interactive user interfaces and why these interfaces are challenging for most description techniques. The state of the art in this field is also introduced in this section. Section 3 presents a development process dedicated to the framework. Then, we use a case study in order to recall the basic concepts of the ICO formalism and the recent extensions added in order to take into account post-WIMP interfaces' specificities. Section 5 presents the case tool PetShop and how the case study presented in the previous section has been dealt with. Lastly, we show how PetShop can be used for interactive prototyping.

### 2 Highly Interactive User Interfaces

Interfaces are usually called highly interactive (or post-WIMP) when the dialogue is directly driven by the user and when the interaction style is direct manipulation [18]. This type of interaction allows the user manipulating domain objects directly i.e. without using standardized, predefined interactors [10] as for WIMP interfaces. Papers dealing with this kind of interfaces usually put the emphasis on the novelty of the interaction style of such interfaces [21] (use of new input devices and new visualization techniques). The parallel and asynchronous nature of their handling of inputs and outputs make their behaviour too complex to be described by means of classical description techniques [11]. Description techniques used for modelling such interaction styles and techniques must be able [3] to take into account existing as well as new applications, to compare various design alternatives and to make easier the identification and the definition of new interaction techniques.

If we consider interfaces such as the ones developed in the field of Air Traffic Control for instance, a new characteristic appears which is the dynamicity of interaction objects in terms of existence, reactivity and interrelations [6]. At the opposite of WIMP interfaces, where the interaction space is predetermined, these interfaces may include new interactors (for instance graphical representations of planes) at any time during the use of the application. Even though this kind of problem is easily mastered by programming languages it is hard to tackle in terms of modelling. This is why classical description techniques must be improved in order to be able to describe in a complete way highly interactive application.

### 3 DEVELOPMENT PROCESS OF INTERACTIVE AND SAFETY CRITICAL SOFTWARE SYSTEMS
The development process of software systems has been the focus of a significant amount of work in the field of software engineering. Early models (including waterfall and V models [12]) focus mainly on the identification and the clear separation of the various phases of the development of software systems. However, the way they represent the process (i.e., in a linear and mainly one-way structure) is very limited and not able to deal with prototyping issues. The spiral development model was introduced by Boehm in 1988 (see Figure 1) to deal explicitly with prototyping and iteration. Prototyping is a key issue in the development of interactive systems and thus this model has been widely adopted.

Figure 1. The Spiral model (from [5])

However, this model does not encompass the various models that have to be built during the development of an interactive system. For instance, it does not refer to task models that are now recognized as critical for the design of usable interactive systems. Besides, usability evaluation is not explicitly represented in the model thus leaving user involvement unsupported and at the discretion of software engineers. Research has been conducted in this field and the star model [8] explicitly introduces explicitly task analysis and usability evaluation as phases of the development process. However, most of the phases must generally be conducted manually i.e., without tool support for both representing and analyzing the models built during those phases. This is not critical when dealing with "classical" (i.e., non-safety critical) interactive systems, but each manual operation may be source of a fatal error when safety critical systems are concerned.

Figure 2. Various phases of the design process of interactive systems on the (left), user related parts (right)

Figure 2 presents the various phases of the development process of a software system with an emphasis on their deliverables. The phases that require human creativity and intervention are represented as clouds on the figure. An important aspect is that each phase produces several models corresponding to the top-down process of taking into account information in a stepwise refinement manner. Even though the process is highly iterative (as for the spiral model) we have decided to represent here only the flow of information between the various phases.

When dealing with interactive systems it is now widely agreed upon that user information (that leads to usability) has to be taken into account and that task analysis and task modelling may support this activity. Right hand side of Figure 2 shows that user goals have to be analyzed and that their
level of abstraction is the same as the one of specification phase, while task analysis level corresponds to design phase.

The advantage of using formal notations to support the design of the models is the potential for mathematical verification they provide. A formal model (whether it describes a task or a system) may be automatically checked for several behavioral properties such as deadlock freedom or termination or other more user related properties such as honesty or predictability [7]. If, at a given stage, the model fails to comply with such properties, it has to be reworked and corrected until the problem is solved.

If formal methods are used during the design process, the coding phase can be at least partly automated for instance by means of code generation. This automation of the coding phase can also be done by the interpretation at run-time of the models built in the earlier phases (this is close to the model-based approaches to the design of user interfaces, see [22], [20]). We have previously investigated the pros and cons of these two approaches in [1].

It is important to notice that such a process may, at the very best, ensure the system under design is "defect free". This process alone can by no means ensure that the system will be usable at all, and much less that it will be "user friendly". In order to cope with all the issues and thus deal both with safety and usability requirements that are crucial for safety critical interactive applications, we propose an iterative development process based on formal notations.

The solution we propose to this end is twofold:

1. We propose a development process supporting the use of formal notations and iterative user-centered prototyping. This process is based both on prototyping and formal description technique in order to deal with reliability and usability concerns. This process is tool-supported as it is shown on the Air Traffic Management case study in next sections.

2. We propose a formal notation for the modeling of interactive systems. This brings the advantages of formal approaches, the most important of which are conciseness, consistency and lack of ambiguity. This also makes task models amenable to mathematical verification i.e. the possibility for proving properties over system models.

![Figure 3. Integrating formal prototyping with "classical" software development](image)

The left-hand side of Figure 3 presents the iterative prototyping phase. This is what we call the super-high fidelity prototyping development phase. Indeed, this is not a low-fidelity prototyping activity as the prototype produced represents the exact look and feel of the final system. It is more than a high-fidelity prototype as the system is not built using a "classical" Rapid Application
Development (RAD) environment (such as Microsoft Visual Basic for instance). Indeed, the system is built using the ICO formalism that is analyzed (through formal analysis) and executed. These prototyping activities allow for taking into account user's constraints and needs through a stepwise refinement process and ensuring a certain quality of the system through the formal verification of the specifications.

It is important to notice that the building of the system is not finished at the end of this prototyping phase. Indeed, development of safety critical applications require a more structured and global development process as the one promoted by the waterfall model. The right-hand side of Figure 3 shows the basics of the waterfall development process. This Figure represents also how the prototyping iterative process and the waterfall one are related. The approach we promote provides several valuable inputs for this "classical" development process:

- a set of validated requirements elicited and tested by the users during the prototyping phase and thus reducing time spent in the requirement phase;
- a set of formal specification of the interactive part of the application. These specifications will be used as inputs in the specification phase and will thus contribute to reduce development time;
- a complete user interface (both its presentation and its behavior) that will have to be re-implemented in the development phases. Indeed, the user interface produced by the execution of the ICO specification cannot be used for the final system as our tool, PetShop, is oriented towards interpretation and thus cannot reach the level of performance required for safety critical applications.

4 A Case Study

The ICO (Interactive Cooperative Object) formalism aims at describing in a formal way the dialog part of an interactive application in order to validate models through specific verification techniques. The formalism uses concepts borrowed to Object-Oriented approaches (such as classification, dynamic instanciation, encapsulation, inheritance and client-server relationship) for describing data or static aspects of the interactive system. Behavioural and dynamic aspects are described using High-Level Petri nets.

4.1 Informal Description of the Example

The example presented in this section comes from a more complex application studied in the context of the European project Mefisto. This project is dedicated to formal description techniques and focuses on the field of Air Traffic Control. This example comes from an En-route Air Traffic Control application focusing on the impact of data-link technologies in the ATC field. Using such applications air traffic controllers can direct pilots in a sector (a decomposition of the airspace). The radar image is shown on Figure 4. On the radar image each plane is represented by a graphical element providing air traffic controllers with useful information for handling air traffic in a sector. In the simplified version of the radar image we are considering, each graphical representation of a plane is made up of three components: a label (providing precise information about the plane such as ID, speed, cleared flight level, …), a dot (representing the current position of the plane in the sector) and a speed vector (a graphical line from the dot which represent the envisioned position of the plane in 3 minutes).

On the top of the window a button named (New Plane) allows for introducing new plane in the sector. This button allows for simulating the arrival of new planes in the sector while in reality they would be instantiated on the user interface by calls from the functional core of the application processing information provided by physical radars.

Initially the radar image is empty. Each time the button is pressed, a new plane is randomly generated. It is possible for the user to select planes by shift-clicking on its graphical representation. A right-click on a selected plane opens a menu made up of two elements (this is only for sake of simplicity): Abort, for closing the menu and Delete, for deleting the plane (removing its graphical representation from the radar image).
At a time only one plane can be selected and the menu can only be opened for a selected plane (on Figure 4 the plane 768 the menu is opened). Planes are automatically removed when leaving the sector.

4.2 Formal Description of the Example

This section presents both the formal specification of the case study introduced in the previous section and the main concepts of the ICO formalism.

Object architecture of the application. The first step in the modelling process is to identify the objects of the application and their relationships. The ICO-based specification of the case study consists in several object classes, pictured in Figure 5 using the UML diagrammatic notation [16]. To model the case study we have identified three classes, as shown by the top part of Figure 5: a class PlaneManager managing all the planes in a sector, a class Plane (one instance of this class is created each time the user clicks on the New Plane button) and a class menu (for which only one instance exists for one Plane instance). The bottom part of Figure 5 pictured the relationship between instances of the three classes, in terms of communication (i.e. method calls).

![Object Structure](image)

Figure 5: The object structure of the case study

Behaviour of the application. For each object of the application, we have to describe the set of services offered by this object to its environment (i.e. the other objects of the application). For this purpose we use the Interface Description Language (IDL) promoted by the Object Management Group in CORBA [15]. Figure 6 contains the IDL description of the three classes.

```idl
module PlaneManager {
    interface IPlaneManager {
        void openPlane(in Plane) ;
        void closePlane(in Plane) ;
    }
}
```
When the IDL is defined for each class it is important to describe more precisely the behaviour of each class. This behaviour defines according to the internal state of each object of a class what services are currently available. It also defines what state changes occur when a service is performed. Next three figures present the behaviour of the three classes modelled by high-level Petri nets.

Figure 7 presents the ObCS of the class PlaneManager. As this class is in charge of creating and managing planes in a sector it is in charge of instantiating objects from the Plane class. This functionality is modelled by the transition newPlane in the model and the reference to the newly created plane is stored in the outgoing place of this transition, i.e. the place Plane.
This is one of the arguments for using high-level Petri nets as they allows for representing in a graphical and formal way dynamic instanciation of objects which is a mandatory requirement for modelling highly interactive applications as introduced in section 2.

Figure 8: ObCS of the class Plane

Figure 9: ObCS of the Class Menu

**widget** PlaneManager

//No rendering method.

**widget** Plane
```java
void show() {
    // Display plane as a graphical point, a speed vector and a label. The plane is shown as unselected.
}
void hide() {
    // Remove plane from the screen.
}
void showSelected() {
    // Show the Id of the plane surrounded by a green rectangle and the dot circled by a green circle.
}

widget Menu
void show() {
    // Display menu.
}
void hide() {
    // Hide menu.
}
```

Figure 10: Rendering methods for the case study

Description of the presentation part. In this step of the specification process, information is added to the behavioural specification for describing the user interface part of the application. Three different elements have to be defined:

? the rendering methods (i.e. for this case study the set of methods dedicated to displaying graphical information on the screen). They are represented in **Figure 10**

? the activation function. As for the behavioural part, for each class an activation function is defined (see **Figure 11**). For each couple (Object, event) this function associates a user service. A user service is defined by a set of transitions in the ObCS. These transitions can only be fired if they are available (according to the current state of the application) and the event has been triggered through user action on the input device. Graphically, each transition of this set is represented in the ObCS with an incoming broken arrow with an oval labelled by the name of the event. Managing dynamically instantiated objects requires the referencing of the objects within the activation function. This is modelled by the inclusion of a variable to which the event is associated (see for instance the definition of the selection of a plane in **Figure 11**) and of course the place in which this information is stored.

? the rendering function aims at representing the reaction of the system when a state change occurs. In Petri nets, a state change can only occur when tokens are removed or added to a place. Thus the rendering function associates, to each of these possible state changes, a previously defined rendering method. **Figure 12** shows for each class and for each place of the ObCS of this class which rendering method is triggered when a token is removed or added to a place.

<table>
<thead>
<tr>
<th>Place</th>
<th>Object type</th>
<th>Event</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>ButtonNewPlane</td>
<td>Click</td>
<td>newPlane</td>
<td></td>
</tr>
<tr>
<td>Planes</td>
<td>Plane</td>
<td>Shift Left Click</td>
<td>select</td>
</tr>
</tbody>
</table>


Selected Plane | Plane | Left Click <x> | deselect

**Plane**

<table>
<thead>
<tr>
<th>Place</th>
<th>Object type</th>
<th>Event</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plane</td>
<td>Right Click</td>
<td>userTryOpenMenu</td>
<td></td>
</tr>
</tbody>
</table>

**Menu**

<table>
<thead>
<tr>
<th>Place</th>
<th>Object type</th>
<th>Event</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>MenuItemAbort</td>
<td>Click</td>
<td>abort</td>
<td></td>
</tr>
<tr>
<td>MenuItemDelete</td>
<td>Click</td>
<td>remove</td>
<td></td>
</tr>
</tbody>
</table>

Figure 11: Activation function of the case study

---

### PlaneManager

<table>
<thead>
<tr>
<th>Element of the ObCS</th>
<th>Rendering method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>State changing</td>
</tr>
<tr>
<td>Place Planes</td>
<td>Token &lt;x&gt; enters</td>
</tr>
<tr>
<td>Place RemovedPlanes</td>
<td>Token &lt;x&gt; enters</td>
</tr>
</tbody>
</table>

---

### Plane

<table>
<thead>
<tr>
<th>Element of the ObCS</th>
<th>Rendering method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>State changing</td>
</tr>
<tr>
<td>Place selected</td>
<td>Token enters</td>
</tr>
</tbody>
</table>

---

### Menu

<table>
<thead>
<tr>
<th>Element of the ObCS</th>
<th>Rendering method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>State changing</td>
</tr>
<tr>
<td>Place Closed</td>
<td>Token enters</td>
</tr>
<tr>
<td>Place Opened</td>
<td>Token enters</td>
</tr>
</tbody>
</table>

Figure 12: Rendering function of the case study

---

**5 Petshop Environment**

In this section we present the PetShop environment and the design process it supports. Some screen dumps are included in order to show what is currently available. **Figure 13** presents the general architecture of PetShop. It is composed of a set of rectangles and documents-like shapes. The rectangles represent the functional modules of PetShop. The documents-like shapes represent the models produced and used by the modules.
5.1 Editing the ObCS

PetShop features an Object Petri net editor that allows for editing and executing the ObCSs of the classes.

All the components of the ObCS can be interactively built using PetShop. PetShop also automatically generates a skeleton Object Petri net from the IDL description [19]. This Object Petri net is not the final ObCS of the class but only the minimal set of places and transitions corresponding to the IDL description.
The editing of the Object Petri net is done graphically using the Palette in the centre 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.

5.2 Editing of the Presentation

Currently, PetShop is linked to JBuilder™ environment for the creation of the presentation part of the ICOs. Thus creation of widgets is done by means of JBuilder interface builder (see Figure 15). This figure shows how the PlaneManager components are created using the interface builder.

5.3 Execution

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 16 shows the execution of the specification of the Plane Manager in Petshop. The ICO specification is embedded at run time according to the interpreted execution of the ICO.

At run time, the user can both look at the specification and the actual application. They are in two different windows overlapping in Figure 16. The window PlaneManager corresponds to the execution of the window with the Object Petri net underneath.

In this window we can see the set of transition that are currently enabled (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 PlaneManager the event is passed onto the interpreter. If the corresponding transition is enabled then the interpreter fires it, performs its action (if any), changes the marking of the input and output places and perform the rendering associated (if any).
5.4 PetShop and MVC

We noted that the edition and the verification of the models are not completely made in an exclusive way, and that the number of round-trip between these two tasks is especially important. To promote the lack of modes, we based PetShop on an architecture that results from the MVC design pattern [13].

MVC (Model - View - Controller) is a design pattern that structures a graphical interactive system into a set of Models, Views and Controllers. On one side, models represent the set of data of the application field that might be displayed, and, on the other side views and controllers provides means to graphically modify (controllers) or display (views) these models. When a controller modifies a model, this model notifies the modification to all its views.

Figure 17 shows the simplified MVC structure of PetShop, where a Petri nets interpreter plays the role of model for four views, one playing the role of controller too.

As an example, Figure 18 shows the edition window of PetShop, divided into four parts (each of them corresponding to a view in Figure 17):
? the top right part is the Petri nets editor that is a both a view and a controller, as it provides means to interact with the edited Petri net,
? the top left part is the IDL interface displayer,
? the bottom right part is the matrix view of Petri nets,
? the bottom left part is the set of properties of the edited Petri net, produced by the Petri net analyser

Figure 18: The MVC structure of PetShop on an example

6 Interactive Prototyping

Within PetShop, prototyping from specification is performed in an interactive way. At anytime during the design process it is possible to introduce modifications either in order to make the specification more precise or to change it. The advantage of model-based prototyping is that it allows designers to immediately evaluate the impact of a modification.

We have identified two different kinds of modifications that can be performed using Petshop, namely : lexical and syntactic. Provided the paper is accepted, for each of these modifications an example will be presented at the conference.

6.1 Lexical Modifications

Lexical part of the user interface gathers elementary elements of the presentation (for instance the drawing of a button) and all the elementary actions offered to the user (such as clicking on a button). Lexical modifications concern the addition, the removal or the modification of this kind of elements.

? Changing the rendering of a plane. When selected, the colour of a plane changes to green. As a lexical modification, we propose to change it to red.

-at the specification level: nothing changes in the specification. Only the content of the method `showSelected` must be modified and this must be done using the Jbuilder environment.
Changing the event triggering the selection of a plane. The currently used event (as defined in 4.1) is Left Button Shift Click. We propose to use instead the event Left Button Click. Therefore we modify:

- at the specification level: the corresponding event must be changed in the activation function.
- at the code level: the Java code must be modified to change the adapter (representing the activation function) of the widget plane.

6.2 Syntactic Modifications

The syntactic part of the user interface describes the links and relationships between the lexical elements (for instance shift clicking on a plane then right clicking on the plane to open the menu and then deleting the plane).

Modification of the selection mechanism. Currently only one plane can be selected at a time. In order to allow multiple selection the following modifications must be performed:

- at the specification level: the inhibitor arc (the arc terminated by a black circle) linking the transition select to the place SelectedPlane (see Figure 7) must be removed,
- at the code level: no modification.

Defining a upper limit for the number of planes in the sector. In the initial informal specification there is no limit on the number of planes. Adding a maximum limit to 20 planes (number of planes normally controlled by a controller) requires the following modifications:

- at the specification level: A new place must be added in the PlaneManager ObCS (Figure 7). Initially this place will hold 20 tokens. This place has to be connected by an arc to the transition NewPlane of the same ObCS. When a plane leaves a sector (or is deleted using the menu) the corresponding transition must add a new token to this place.
- at the code level: no modification.

7 Conclusion

Prototyping is now recognised as a cornerstone of the successful construction of interactive systems as it allows making users at the centre of the development process. However, prototyping tends to produce low quality software as no specification or global design is undertaken. We have shown in this paper how formal specification techniques can contribute to the development process of interactive systems through prototyping activities.

While the ICO formal specification technique has reached a maturity level allowing coping with real size dynamic interactive applications, the Petshop environment is still under development. A real size application has been completely specified in the field of the European project MEFISTO (http://giove.cnue.cnr.it/mefisto.html). More information about PetShop can be found on PetShop's web site: http://lis.univ-tlse1.fr/petshop/

However, the work done on this Air Traffic Control application has also shown the amount of work that is still required before the environment can be used by other people than the ones that took part in its development.

In order to make it attractive to developers we are integrating additional features such as:

- tool-supported verification of properties,
- analysis of conformance with other representations such as tasks models
- performance analysis in order to support widgets selection and structuring

Another stream of research we are investigating is the generation of test cases from the formal specification in order to help developers checking whether an implementation is conformant with respect to the specification. This will allow development teams to take the specifications, still use their favourite programming environment and later check whether their implementation is conformant with it. This work is more mature in the field of distributed applications [2] but reusing it for interactive dynamic applications remains to be done.

8 References


