

OP: A Novel Programming Model for Integrated Design and Prototyping of Mixed Objects

Céline Coutrix, Laurence Nigay

▶ **To cite this version:**

Céline Coutrix, Laurence Nigay. OP: A Novel Programming Model for Integrated Design and Prototyping of Mixed Objects. INTERACT 2011 - International Conference on Human-Computer Interaction, Sep 2011, Lisbon, Portugal. pp.54-72, 10.1007/978-3-642-23765-2_5 . hal-00758552

HAL Id: hal-00758552
https://hal.science/hal-00758552

Submitted on 28 Nov 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

OP: A Novel Programming Model for Integrated Design and Prototyping of Mixed Objects

Céline Coutrix, Laurence Nigay

CNRS, Université Joseph Fourier Grenoble 1
Laboratoire d'Informatique de Grenoble LIG UMR 5217, Grenoble, F-38041, France
{Celine.Coutrix, Laurence.Nigay}@imag.fr

Abstract. In the context of mixed systems that seek to smoothly merge physical and digital worlds, designing and prototyping interaction involves physical and digital aspects of mixed objects. However, even though mixed objects are recurrent in the literature, none of the existing prototyping tools explicitly supports this object level. Moreover, designers have to use distinct tools, on the one hand, tools for designing ideas and on the other hand tools for prototyping them: this makes the design process difficult. To help alleviate these two problems, we present OP (Object Prototyping), a toolkit that provides a new programming model focusing on mixed objects and allows us to seamlessly go back and forth from conceptual ideas to functional physical prototypes, making the iterative design process smooth and integrated. Indeed, OP is explicitly based on an existing conceptual design model, namely the Mixed Interaction Model that has been shown to be useful for exploring the design space of mixed objects. Our user studies show that, despite its threshold, designers and developers using OP can rapidly prototype functional physical objects as part of a design process deeply intertwining conceptual design with prototyping activities.

Keywords Prototyping, Toolkit, Mixed Systems, Mixed Objects, Augmented Reality, Physical User Interfaces, Tangible User Interfaces, Design.

1 Introduction

Mixed interactive systems seek to smoothly merge physical and digital worlds. Examples include tangible user interfaces, augmented reality, augmented virtuality, physical interfaces and embodied interfaces. In mixed systems, users interact with objects existing in both the physical and digital worlds. These objects are depicted in the literature as augmented objects, physical-digital objects or mixed objects.
Nowadays, designers either work alone or work with developers to design such objects. In the first case, except if they have coding skills, they cannot produce interactive mockups. Some tools provide partial contribution *towards* this problem, like Intuino [26]. In the second case, they can either provide a description to a developer who will then develop a prototype (however this significantly slows the design process) or they can also work together with a developer in designing and prototyping. This latter case is the work practice that we target in our study. In this context, facilitating the

systematic exploration of design solutions is a very important problem for designer-developer pairs, as their aim is to generate solutions in the exploration phase [4][16].
In this paper, we introduce OP - Object Prototyping - a toolkit for the prototyping of such objects involved in mixed systems. OP offers two main benefits for designer-developer pairs: (1) it offers a new programming model, based on interacting objects and (2) it provides an integrated design approach, thanks to its strong link with an interaction model that was proven useful for exploring the design space. These two key aspects of OP are motivated by the following problems.

First, several studies of mixed systems developed in the literature [25][7][12] underline that the concept of a mixed object is central and recurrent in the design of mixed systems. A mixed object can be either used as a tool by the user to complete her task like the paper button "fill" in the drawing scenario of the seminal Digital Desk [27], or as the object focus of a task like the drawn house whose roof is to be filled in the drawing scenario of the Digital Desk. Unless they explicitly focus on objects in mixed systems, prototyping toolkits would not provide an optimal abstraction level for better design flexibility. Moreover reuse could be improved, as a mixed object can be an elementary block reused in different applications, like the pucks of [21], a generic mixed object.

Second, as the aim of early design is to explore as many ideas as possible [16][4], researchers and practitioners have proposed: (1) Interaction models, *e.g.* [14][7][9], in order to help to systematically explore the design space, (2) Prototyping toolkits, *e.g.* [13][11], in order to help materialize design ideas and support active thinking.

The use of both types of tools has been shown as being effective [7][11]. Moreover conceptual design and prototyping activities are inextricable [16]. Both activities inform each other: designer-developer pairs are going back and forth from conceptual ideas to practical realizations. An optimal practice of intertwined conceptual design and prototyping would be to explore the design space based on a model while prototyping the design alternatives. However, interaction models and prototyping toolkits are hardly used together. These two types of tools mainly remain unrelated, making their simultaneous use difficult. By using either one of these types of tools, designer-developer pairs therefore end up with either (a) a good covering of the design space, but cannot experience their design ideas with prototypes, or (b) prototypes that they can reflect on or show to stakeholders but cannot be sure that they explored the design space in a satisfactory way and might have forgotten to consider interesting solutions. As pointed in [2], after the presentation of an interaction model for GUI, "operationalizing the design of interaction requires appropriate tools and frameworks."

To help alleviate both of the two above problems in the design of mixed physical-digital systems, OP is directly based on a conceptual interaction model that focuses on mixed objects. As OP tightly integrates the widely used Qt [23], the Mixed Interaction Model and existing hardware tools to offer a structuring of the prototype, it allows designer-developer pairs to easily go back and forth from ideas to prototypes of a mixed object.

The remainder of this paper is organized as follows: we begin by recalling the key elements of the Mixed Interaction Model, that is the background of this work. We then explain to which extent existing tools address this practice of intertwined conceptual design and prototyping of mixed objects. We next present the OP prototyping toolkit and we end by describing experiences of designer-developer pairs using our tool.

2 Background: the Mixed Interaction Model

The Mixed Interaction Model (MIM) is a conceptual design model that focuses on hybrid physical-digital objects or mixed objects. Some interaction models for GUI [2] also identified this intermediary level between hardware resources and applications that is called the object level. The key contribution of MIM is the structuring of a mixed object: MIM defines building blocks of a mixed object at a higher level of abstraction than an encapsulation of a piece of hardware. The MIM structuring of a mixed object has been shown to be useful for exploring the design space of mixed objects in a systematic way [7].

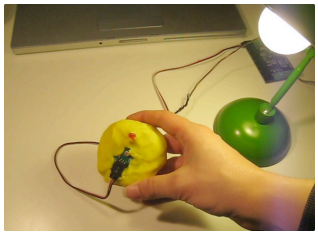


Fig. 1. A mixed physical-digital object sensitive to light that has been built with OP.

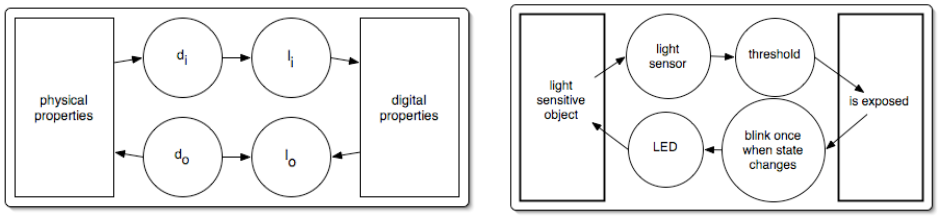


Fig. 2. (a) Model of a mixed object. (b) Modeling of a simple light sensitive object (Fig.1).

Based on MIM, a mixed object is defined by its physical and digital properties as well as the link between these two sets of properties. The link between the physical and the digital parts of an object is defined by linking modalities. The definition of a linking modality is based on that of an interaction modality [3]: Given that d is a physical device that acquires or delivers information, and l is an interaction language that defines a set of well-formed expressions that convey meaning, an interaction modality is a pair *(d,l)*, such as *(camera, computer vision)* or *(microphone, pseudo natural language)*. These two levels of abstraction, device and language, are reused. But as opposed to interaction modalities used by the user to interact with mixed environments, the modalities that define the link between physical and digital properties of an object are of lower abstraction level and therefore called *linking modalities*. Fig. 2a shows the two types of linking modalities that compose a mixed object: An input linking modality (d_i,l_i) is responsible for (1) acquiring a subset of *physical properties*, using a device d_i (input device), (2) interpreting these acquired physical data in terms of *digital properties*, using a language l_i (input language). An output linking modality is in charge of (1) generating data based on the set of *digital properties*, using a language l_o

(output language), (2) translating these generated physical data into perceivable *physical properties* thanks to a device d_o (output device).

To illustrate the different parts that compose a mixed object, the modeling of a simple object, a light sensitive object shown in Fig. 1, is provided in Fig. 2b. It embeds a light sensor and a LED on its surface. Each time the user sets the object too close to a light source, the LED blinks once. Then, if the user moves the object far enough from the light source, the LED blinks once again. Fig. 2b shows the modeling of this object: The light sensor (input linking device) captures the level of exposure. If this level is above a threshold (input linking language), then the digital property, `is exposed`, is set to true. If the level falls under the threshold, the digital property is set to false. Each time the state of the digital property changes, an output linking language `beep` is activated. This output linking language triggers the LED to blink once.

3 Related Work

The existing toolkits for mixed systems could be analyzed based on their type of underlying language (compiled vs. interpreted, textual vs. graphical, etc.), their threshold and ceiling [18], their popularity in use, etc. We take a different viewpoint on existing tools by analyzing (1) how these tools support the structuring of the prototype based on a mixed object and (2) to which extent they support a systematic exploration of the design space by relying on the ability of an interaction model to help designers create new designs [2].

3.1 The mixed object programming model

Phidgets [22], Arduino (http://arduino.cc/), ARToolKit [1], Intuino [26], BOXES [13], MaxMSP [17] and PureData (http://puredata.info/) do not imply a code structuring based on a mixed object in their resulting code. For instance, Fig. 3 shows a Phidgets C code for the light-sensitive object of Fig. 1 and 2b. Compared with the MIM modeling of Fig. 2b, the code corresponding to the linking input language is scattered at lines 3, 4, 6, 8 and 10, interwoven with code for input and output devices and the output language. As a consequence, it is very difficult to localize all the lines of code corresponding to a particular element of a mixed object (principle of separation of concerns), as identified in the MIM modeling of Fig. 2b. Arduino has a similar approach with setup() and loop() functions resulting in a code structured in a different way. ARToolKit offers a similar loose structuring of the code, resolving difficulties of programming with camera and computer vision techniques, and not difficulties of programming with sensors.

Unlike Phidgets, well-known in the research community, MaxMSP is on the contrary widely used in the art/design community. Like its open-source counterpart PureData, it is a graphical programming language (example in Fig. 4). However, as such, it does not necessarily guide the resulting prototype to follow a particular structure. The patch shown in Fig. 4 is one code out of numerous possibilities for the light sensitive mixed object (Fig. 1). We designed this solution to be as simple as possible but also as close as possible to the MIM modeling of Fig. 2b. However we

observe that the portions of the code corresponding to the linking input device (1 in Fig. 4), the linking input language (2 in Fig. 4) and the linking output device (5 in Fig. 4) are mixed together. Moreover, while some elements of a MIM modeling, like the ones mentioned above, are composed of boxes (objects: boxes with borders, messages: boxes without borders) and connectors, others like the linking output language (4 in Fig. 4) is composed of a single connector, therefore making it difficult to visually localize the blocks composing the code in regard to a MIM modeling.

```
#include <phidget21.h>
CPhidgetInterfaceKitHandle Kit;
int OldValue = 0;
int SensorChangeHandler(CPhidgetInterfaceKitHandle h, void *p, int i, int v)
{
 if ((v < 500 && OldValue >= 500) || (v > 500 && OldValue <= 500)
 {CPhidgetInterfaceKit_setOutputState(Kit, 0, 1);}
 else
 {CPhidgetInterfaceKit_setOutputState(Kit, 0, 0);}
 OldValue = v;
 return 0;
}
int main (int argc, char * const argv[]) {
 CPhidgetInterfaceKit_create(&Kit);
 CPhidgetInterfaceKit_set_OnSensorChange_Handler (Kit, SensorChangeHandler, NULL);
 CPhidget_open((CPhidgetHandle)Kit, -1);
 int result;
 const char* err;
 if((result = CPhidget_waitForAttachment((CPhidgetHandle)Kit, 10000)))
 {CPhidget_getErrorDescription(result, &err);
  printf("Problem waiting for attachment: %s\n", err);}
 while (true) {}
}
```

Fig. 3. Phidgets C code of the light sensitive object presented in Fig. 1 and modeled in Fig. 2b.

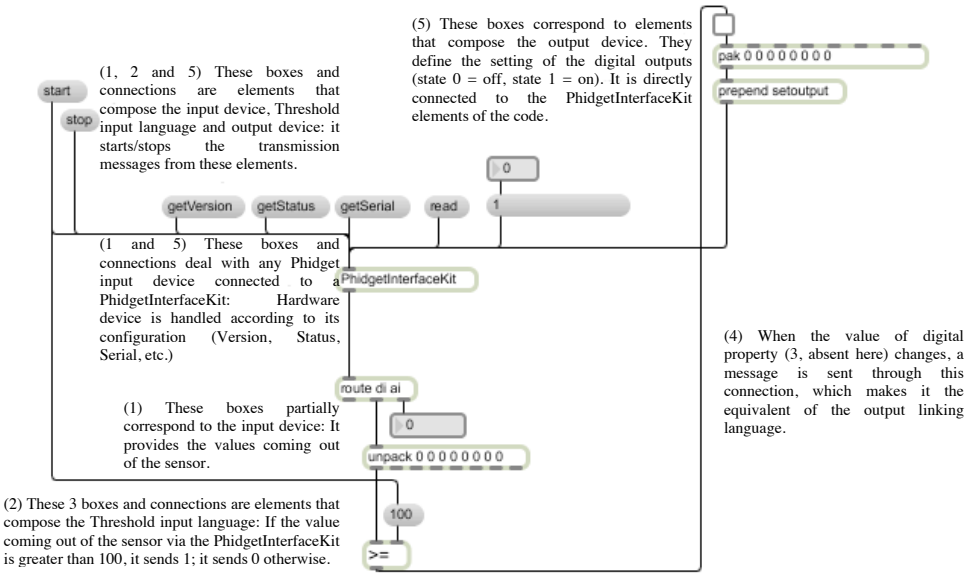


Fig. 4. MAX/MSP patch of the light sensitive object presented in Fig. 1 and modeled in Fig. 2b.

ICON [8], ICARE [3], OpenInterface [20] and Context toolkit [24] identify the object level, but at a higher level of abstraction that does not allow its design. Indeed, with those toolkits, the object is a predefined brick used in the design of an entire application. Moreover, they mostly focus on interaction from the user to the application and hardly consider the feedback towards the user.

Papier-mâché [15] and d.tools [11] are the closest to the object programming model. On the one hand, papier-mâché allows the designer to manipulate the physical part (called Phob) and the digital part (called noun or verb), but does not use a detailed definition of the association between those two parts. Moreover, it allows connecting a physical part to "noun", equivalent to digital properties, but also to "verb" equivalent to commands. The command associated with a tool is application-dependant and is at a higher level of abstraction than the object level. On the other hand, d.tools implies objects loosely structured according to physically stuck elements rather than conceptual objects. This approach can possibly combine an object that is a tool with a task object, as a unique entity. Instead of focusing on the design of mixed objects, it considers the design of a mixed system as a whole.

We now examine to which extent existing tools support a systematic exploration of the design space and are related to a conceptual model.

3.2 The systematic exploration as a design approach

Several toolkits have no explicit link to an interaction model. Among those, Phidgets [22], Arduino, ARToolKit [1], MaxMSP [17], PureData, Context toolkit [24] and ICON [8] are based on existing developers' practices and solve technological problems, not design problems. They either provide textual coding (Phidgets, Arduino, ARToolKit, Context toolkit) or graphical user interface (GUI) for coding (MaxMSP, PureData, ICON) but do not promote exploration of alternatives according to an interaction model. For instance, ARToolKit and Phidgets use callback functions for hardware resources. As a consequence, coding/development and design practices are not close enough to allow designers to easily go back and forth from design to prototyping with these toolkits without restructuring their ideas to fit the programming model.

BOXES [13], d.tools [11] and Intuino [26] are not based on developers' practices, but on non-instrumented design practices. Intuino provides a GUI to better fine-tune sensors and actuators signals used with Arduino. The tool BOXES provides a direct link between physical material and an existing GUI. As opposed to BOXES, the toolkit d.tools provides indirect prototyping of the interaction through state transition diagrams, where states define an on-screen graphical representation of the connected tangible object to be prototyped. However, it does not encourage the systematic model-based exploration of alternatives other than the opportunistic ideas that a designer might have.

Papier-mâché [15], ICARE [3] and OpenInterface [20] toolkits have an explicit link to existing interaction models. They support an exploratory design process by proposing a structuring of the code, to be entered either textually like Papier-mâché or graphically like ICARE and OpenInterface. For instance, Papier-mâché introduces associations between physical Phobs and digital elements based on name (object) or verb (command) and as such the tool can be closely related to Fishkin's taxonomy of

tangible user interfaces (TUI) [10]. ICARE and OpenInterface are related to a model of multimodal interaction [3]. However, the conceptual models behind these toolkits are limited to explore the design space of mixed systems [7]. For instance, the concepts of *name* and *verb* convey respectively the concepts of *task object* and *tool* [9] and correspond to two different types of metaphors that can be applied [10], but leave apart all the possibilities for linking modalities.

Table 1. Related works summary.

Underlying interaction model \ Object level	No	Partial	Yes
No	Phidgets, Arduino, ARToolKit, MaxMSP, PureData, Intuino, BOXES	Context toolkit, ICON, d.tools	
Partial		Papier-mâché, ICARE, OpenInterface	
Yes			

Table 1 shows a summary of existing toolkits according to our two axes of analysis: (1) prototyping at the object level and (2) systematic exploration of the design space based on an interaction model.

In order to both support an object programming model as well as a systematic exploration of the design space, we introduce the OP toolkit that is based on the Mixed Interaction Model and capitalizes on these existing tools when possible. By adopting this approach, our purpose is first not to reduce the technological difficulties encountered when building mixed objects. As explained previously, there are toolkits that answer these problems such as computer vision toolkits (ARToolKit, Papier-Mâché) or hardware toolkits (Phidgets, Arduino). Our toolkit has to be built upon them and be able to integrate the ones resolving technological challenges. OP consequently provides lower thresholds than low-level toolkits on top of which OP is built on. Second the OP toolkit focuses on mixed objects only. OP is not intended as a tool to build an entire application but only mixed objects that are the focus of the design process, as opposed to d.tools, ICON, ICARE and OpenInterface. The OP toolkit could then be included in such tools.

4 OP Toolkit

The OP toolkit offers an extensible library of different types of components based on the Mixed Interaction Model [7]. The toolkit includes around 8000 lines of code providing input/output linking device components, input/output linking language components, composition components for combining devices or languages, as well as digital property components. By doing so, the OP toolkit provides: (1) Modularity at the mixed object level in order to make the interface *flexible* and to make mixed objects *reusable* for other interaction contexts; (2) Modularity at the linking modality level – modules for input and output devices, languages, and compositions – in order to make

mixed objects *flexible* and to make linking modality components *reusable* for other mixed objects; (3) Extensibility in order to make the toolkit itself *flexible*. The developer should be able to add new building blocks and extend to new technologies. In its current version, OP still requires some basic notions of programming for assembling the various parts of a mixed object and for connecting objects to applications. Thus, as explained in the introduction section, the target end-users are designer-developer pairs, developers or designers with some programming skills.

4.1 Building mixed objects

OP currently includes the following components for building mixed objects.

The linking device components are based on three different existing toolkits:

- `VideoInputDevice`, based on the ARToolKit: It captures the video input. Its outputs are images from the input video.
- `MIDIDevice`: This component corresponds to either an input or an output device, and captures/delivers data from/to Interface-Z MIDI sensors/actuators (http://www.interface-z.com/).
- `PhidgetInterfaceKitDevice`: Either an input or an output device, this component captures/delivers data from/to sensors/actuators plugged to a Phidget Interface Kit.

Commonly used devices like speakers or screens are directly supported by the toolkit by linking language components, since computers already support them. So no device components are provided for these standard devices in the current version.

For linking languages, OP offers 10 components. Components amongst them are:

- `IdentityLanguage`: This component does not deform the input values coming from the sensor. A property of the component allows opposing of the output values to the input values. For example, if a sensor provides values between 0 and 999, the output values are then from 999 to 0.
- `RampLanguage`: This component generalizes the `IdentityLanguage` component and implements a deformation of the input values according to a ramp function (see Fig. 5a).
- `ThresholdLanguage`: This component delivers a boolean value true/false if the input integer value is above the threshold and false/true otherwise. Properties of the component are the threshold and a property specifying whether the output value is true above or below the threshold.
- `RepeatLanguage`: Either an input or an output language, this component repeats its inputs several times at specified intervals. Properties are the number of repeats and the interval between repeats.
- `BeepOutputLanguage`: This component corresponds to an impulse function that triggers a beep if used with a sound file or a blink if connected to a LED.

The other five linking language components are documented at http://iihm.imag.fr/demo/op/. In order to prototype more complex linking languages, these linking language components can be connected in series. For example, for making a LED blink twice, we connect a `RepeatLanguage` to a `BeepOutputLanguage`. Moreover for combining linking modalities, we developed

a `Complementarity` component that can combine data coming either from device or language components. For example, for combining the data coming from two accelerometers (Fig. 9g), we use the `Complementarity` component. This component draws directly upon ICARE components [3] for fusion of interaction modalities in multimodal interaction.

For digital properties, OP includes a `DigitalProperty` component that can handle any type of digital property, based on the generic type QVariant from Qt [23].

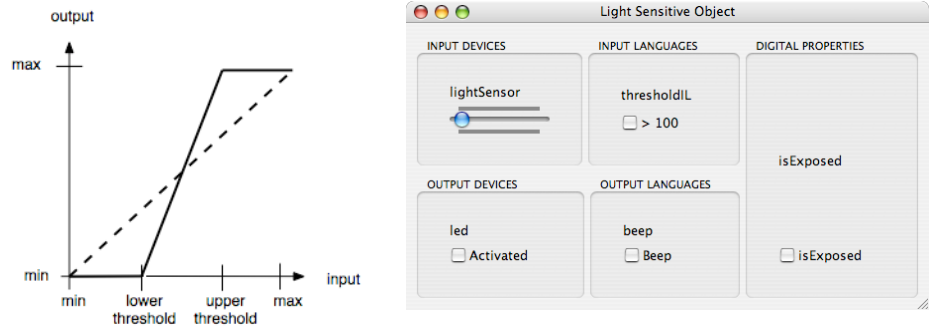
All components, regardless of type, can be connected through the signal/slot mechanism of Qt [23]: A component output is a signal, and a component input is a slot. We connect an input to an output thanks to a line of code: `connect(Component1, SIGNAL, Component2, SLOT)`. OP components have predefined `updated` signals emitted each time its output value is modified, and `update` slots to be connected to the former signals.

Fig. 5c illustrates the use of the library for prototyping the simple light sensitive object of Fig. 1. Using OP components described above, the developers/designers can prototype the object described in Fig. 2b with a few lines of code (Fig. 5c). We use a light sensor from Phidgets in line 3. We set the properties of this component: its name is "lightSensor", its direction is "in", it is plugged to the first input of the circuit board, and it is an analogue sensor (on the contrary to switches for example). According to the modeling of the object (Fig. 2b), we then use in line 6 a threshold component that outputs true when the value is above 100. Line 7 shows how we declare the `isExposed` digital property, with a boolean type. A `BeepOutputLanguage` is used in line 9, without any sound file. Finally a LED connected to the first digital output on the same Phidgets interface kit is declared at line 11. From line 12 to line 19, we connect these five components together, so that outputs of each component provide inputs to the following one. Compared to Phidgets (Fig. 3) and MaxMSP (Fig. 4), the code structure follows the MIM modeling. From this code of a mixed object, it is therefore easy to make modifications. For example:

- Modification of the output behavior of the object: We can make the LED blink four times in order to provide insistence instead of bare observability. Fig. 6 shows how we added a `RepeatLanguage` component and connected it to the other components.
- Modification of the manipulation of the object: We can change the Phidgets light sensor by a pressure sensor from Interface-Z. Fig. 7 shows how we changed the Phidgets light sensor by an Interface-Z MIDI pressure sensor. In this latter case, there is no need to reconnect components, because we only changed the type of the device component, which has no influence on connections.

For fluid design as pointed for the Phidgets or in [16], the toolkit also provides an optional graphical user interface (GUI) to debug/test the prototyped mixed objects. The spatial structuring of the GUI is explicitly based on the MIM interaction model. Fig. 5b shows the graphical tool for the case of the light sensitive object of Fig. 2b. Boxes represent MIM component types. This interface helps the developer/designer to observe and control the details of the behavior of the object. To check at runtime the behavior of the components, developers either act on physical properties (e.g.,

put/move the object close/away to/from a light source) or act on widgets (e.g., move the sliders of Fig. 5b using the mouse). This graphical utility is also used as a wizard of oz tool, in the case of non available components. This enables rapid and early corrections of the prototype.



(a) **(b)**

```
int indexOnBoard = 0;
bool isAnalogue = true;
PhidgetInterfaceKitDevice lightSensor("lightSensor", LinkingComponent::IN,
indexOnBoard, isAnalogue);

int threshold = 100;
bool isTrueAbove = true;
ThresholdInputLanguage thresholdIL("thresholdIL", threshold, isTrueAbove);

DigitalProperty isExposed("isExposed", QVariant::Bool);

char* soundFile = "";
BeepOutputLanguage beep("beep", soundFile);

isAnalogue = false;
PhidgetInterfaceKitDevice led("led", LinkingComponent::OUT, indexOnBoard,
isAnalogue);

QObject::connect(&lightSensor, SIGNAL(updated(int, QTime)), &thresholdIL,
SLOT(update(int, QTime)));
QObject::connect(&thresholdIL, SIGNAL(updated(QVariant, QTime)), &isExposed,
SLOT(updateProperty(QVariant)));
QObject::connect(&isExposed, SIGNAL(PropertyUpdated(QVariant)), &beep,
SLOT(update(void)));
QObject::connect(&beep, SIGNAL(updated(bool)), &led, SLOT(UpdateOutput(bool)));
```

(c)

Fig. 5. (a) Two input linking language components: Identity (dashed line) and Ramp (plain line). (b) Graphical User Interface. (c) Code with inserted picture of a simple object built using OP.

```
int repeatsNb = 4;
float interval = 0.5;
RepeatLanguage repeat("repeat", LinkingComponent::OUT, repeatsNb, interval);
QObject::connect(&isExposed, SIGNAL(PropertyUpdated(QVariant)),
                 &repeat, SLOT(update(QVariant)));
QObject::connect(&repeat, SIGNAL(updated(bool)),
                 &beep, SLOT(update(void)));
```

Fig. 6. Making the LED blink four times by adding a `RepeatLanguage` component.

```
int indexOnBoard = 0;
bool isAnalogue = true;
int resolution = 7;
MIDIDevice pressureSensor(“PressureSensor”, LinkingComponent::IN, indexOnBoard,
isAnalogue, resolution);
```

Fig. 7. Changing the light sensor by a pressure sensor: the first lines of code are modified.

In this section, we explained how we can build mixed objects using OP components. We now present how we insert such objects into a complete application. We recall that a mixed object in the context of a complete application is either a tool used by the user to perform her/his task or the object that is the focus of the task [7].

4.2. Inserting mixed objects into an application

The toolkit provides modularity at the mixed object level so that an object can be reused and adapted for various interaction contexts or applications. For a mixed object to communicate with an application, the `DigitalProperty` component serves as an interface with the application. The signals and slots of a digital property are connected to the rest of the application: the value of a digital property can be modified by an application through a slot and a digital property can communicate with an application by emitting a signal when its value changes. If the application is written in Qt, connecting the object to the application is straightforward. We only need to write one line of code connecting the signal of the `DigitalProperty` component to the desired slot of the application. A survey in our lab showed that out of 70 responses, 19 persons are familiar with Qt. Therefore, even if Qt is actually used, it is obviously not the only solution in use. For an application not written in Qt, developers only need to define a simple QObject whose slot calls the desired function of the application. This slot must then be connected to the signal of the mixed object. For example, for inserting the tool object of Fig. 1 into Google Earth, we build a simple QObject with the slot presented in Fig. 8.

```
void KeySimulationInteractionLanguage::update(QVariant pMessage){
 if (pMessage.canConvert(QVariant::Bool)){
 if (pMessage.toBool()){ CGPostKeyboardEvent(‘d’,(CGKeyCode)0x02, true);}
 else {CGPostKeyboardEvent(‘d’, (CGKeyCode)0x02, false);}}}
```

Fig. 8. Slot written for the simple light sensitive object in order to work in Google Earth for rotating the earth. CGPostKeyboardEvent is a function from the Apple ApplicationServices framework that allows simulating keyboard events.

When this slot receives true, it simulates a ‘d’ key press, and when the slot receives false, it simulates a ‘d’ key release. For this, we use a function from the Apple ApplicationServices framework that allows simulating keyboard events. Equivalent exists for the other platforms. Once this `update` slot is connected to the signal `PropertyUpdated` of the `isExposed` digital property (Fig. 5c), the light sensitive object allows the user to rotate the earth, each time the object is close to a light source. As another example, Fig. 9h shows the cardboard music controller that was used as an illustration of the BOXES toolkit [13], that we rebuilt using OP: the OP cardboard music controller is connected to the iTunes application (play/pause by pressing the button) in the same way as described above. In this stage of our work, the link to an

application is done by hand, but as soon as OP is integrated in a UIMS like the one described in [20], this link will be embedded within the integrated environment.

Fig. 9 shows a variety of objects that have been prototyped using the OP toolkit, as a demonstration like in [22]. These examples cover a wide range of interaction styles that can take part in mixed systems, from augmented reality to augmented virtuality and tangible user interfaces. In the following section we present user studies, including the evaluation of the mixed object programming model, by detailing the case of one of these designed objects (Fig. 9a).

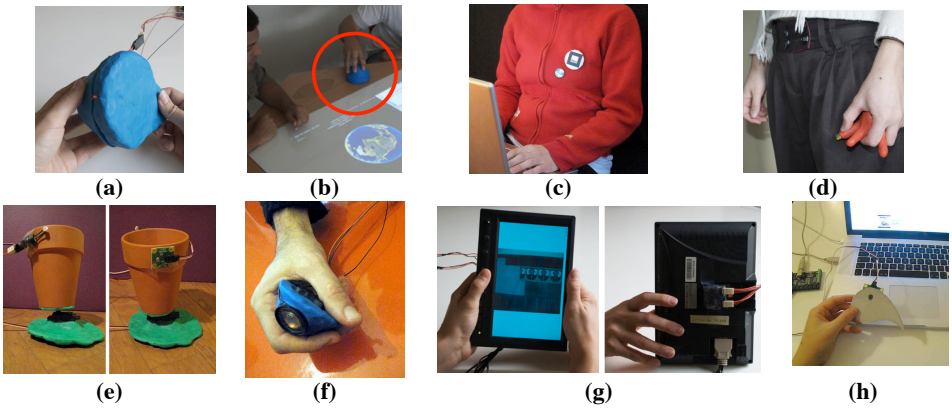


Fig. 9. A selection of mixed objects built with the OP toolkit: (a) Prototype of the ORBIS turnable tool with LED. (b) The ORBIS turnable tool in use on a table for rotating the earth in Google Earth. (c) Prototype of a simple augmented button badge with ARToolKit Marker that detects the presence of a laptop user. (d) "Octopus" tool with foldable tentacles, micro and diodes designed for recording audio [7]. (e) Flowerpot prototype that is oriented with a servomotor according to its most sunny side, thanks to interface-z and Phidgets light sensors. (f) Tool for shuffling pictures in a slideshow, with accelerometers and a loudspeaker: the user shakes it three times and hears a sound as a feedback. (g) Prototype of the ORBIS picture viewer. Pictures are always correctly displayed according to the orientation of the screen: the prototype is based on combined accelerometers fixed on the back side of the screen. (h) The remote control for the music player of [13] that we rebuilt with OP and iTunes.

5 Evaluation and use

Evaluating a software tool is a difficult problem as stated in [19]. As a consequence, the contributions of several software tools like MaxMSP, Arduino, PureData are not evaluated. Other tools, like Phidgets, do not follow requirements advised in [19] for evaluation.

For the OP toolkit, its use in different projects (Fig. 9) is one form of empirical evaluation. Moreover, using OP, it is possible to prototype objects that were developed with other toolkits, like the cardboard music controller [13] (Fig. 9h). To better assess the benefits and limitations of OP, we considered its two key contributions independently. By doing so, the first study aims at evaluating the OP programming

model centered on mixed objects. The second study aims at evaluating the expressive match of the toolkit in the context of an integrated use with the Mixed Interaction Model.

5.1 Evaluation of the mixed object programming model

The OP programming model is based on the definition of a mixed object of the Mixed Interaction Model. This model has shown its benefits for systematic exploration of design alternatives [7]. Therefore, we will not evaluate in this paper the help that the model provides to explore the design space. Nevertheless to further evaluate the OP programming model, we consider the practical problem of designing a physical-digital object that serves as a tool in an application called ORBIS [6]. Consistent with the target users of OP, a typical pair of designer-developer collaborated to design and prototype ORBIS. The system was designed in order to provide new ways to access and enjoy personal pictures, music and videos. ORBIS is to be used in a private, personal and mobile context. In the first version of ORBIS, we only considered tasks related to a list of pictures and Fig. 9g shows a prototype. In this paper, we present the design of a mixed object (i.e., a tool) for navigating the list of pictures in ORBIS. Besides prototyping and testing the OP toolkit, our aim was the joint exploration of design dimensions like appearance and interaction within a systematic approach at the early stage of the design. Before explaining the design steps using OP and the benefits of its programming model, we present the resulting designed prototype, a turnable tool presented in Fig. 9a and modeled in Fig. 10a. For navigating in the list of pictures, the user rotates the tool. The physical angle updates an `angle` digital property through a (potentiometer, identity) linking modality. Another digital property, `level`, is materialized through a (beep, LED) linking modality. Within the ORBIS application, the `angle` digital property is used to compute the index of the current displayed picture. When this index is changed by ORBIS, it modifies the `level` digital property of the tool as a feedback. Since ORBIS has been developed with Qt, the connection between the tool and the ORBIS list of pictures is straightforward. We now explain how the design space has been explored in a systematic way using OP, for obtaining this prototyped tool.

Physical Properties: The design space of the physical shape of the mixed object depends on the appearance (e.g., size, color) and interaction (e.g., affordances). For prototyping the object, we considered the interaction and the appearance in concert, but we only discuss the interaction here. Amongst others, we designed the shape presented in Fig. 9a. From this shape, we studied intrinsic affordances: we found that this shape afforded pressing and turning. Hence we decided that the possible sensed physical properties are pressure and angle.

Input Linking Modality: Because the shape affords pressing and rotating, we prototyped the tool with linking devices that sense angle and pressure. Prototypes with Interface-Z MIDI atmospheric pressure sensor mounted on a balloon and potentiometer are presented in Fig. 11b and c. Since these devices can be connected to the same plug of the Interface-Z circuit board, we did not need to modify the OP linking device component. We experimentally found that pressing was tiring more rapidly than rotating. This draws attention to the fact that even if the design space allows exploration of possibilities, prototyping is essential to assessing a design choice.

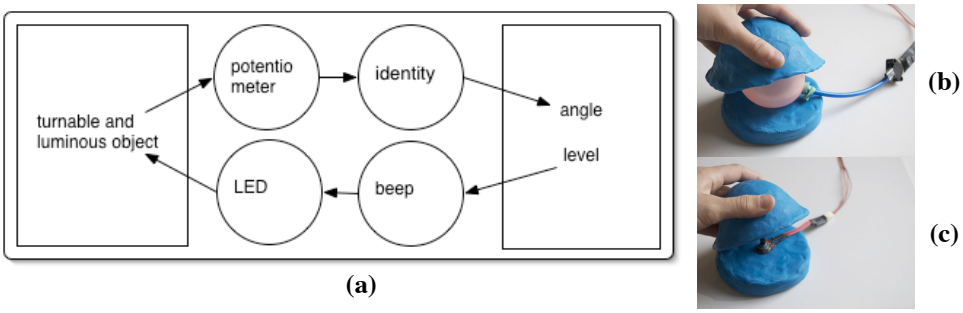


Fig. 10. Modeling of the ORBIS turnable tool presented in Fig. 9a (a) and prototyping the object with an input linking modality sensing (b) pressure or (c) rotation.

The design space of the input linking language is restricted by the choice of the potentiometer and the digital property `angle`. Yet, we envisioned two possibilities: the linking language can take the full circle defined by the shape of the physical object into account. But the gesture for completely rotating the object is not easy to perform. On the contrary, the linking language can only take a subinterval of the full circle into account. We prototyped these two design solutions with different linking languages (Fig. 11): the first solution using an identity language component and the second one using a ramp language component. Tuning the language component using the provided OP components only implies 3 lines of code: the component declaration (the code of Fig. 11a versus the code of Fig. 11b) and the new connection since the name of the component is changed. The same change in an Arduino code costs more, because the programming is not at the object level (Fig. 12a). First, it is not as easy to locate the proper place in the code to insert the equivalent of the input language. Then, one has to add 11 lines of code including the hand-computed transformation (Fig. 12b). The abstraction level of OP is a benefit in this design situation.

(a)
```
IdentityInputLanguage identity("identity", isOpposite, min, max);
```

(b)
```
int lowerThreshold = 42;
int upperThreshold = 84;
RampInputLanguage ramp("ramp", isOpposite, min, lowerThreshold,
upperThreshold, max);
```

Fig. 11. Prototyping the ORBIS object with an input linking language: (a) identity or (b) ramp.

While testing the two resulting OP prototypes, we noticed that even if the ramp language enables users to navigate the range in a small and single movement, it was not easily understood. Hence we chose the identity transformation, even if users have to make several movements in order to turn the tool from 0 to 360 degrees.

Output Linking Modality: In order to materialize the `level` digital property that provides feedback, we designed an output linking modality. We wanted this feedback to be peripheral since it defines the reaction of the tool, while the user focuses on the list of pictures. Towards this purpose, a visual or audio beep was considered as sufficient in comparison with displaying the index of the current picture on top of the picture. As a corresponding device for the chosen language, we could use a LED or a loudspeaker. We developed the two solutions (Fig. 13 and Fig. 14 respectively). The difference between the two pieces of code is that the loudspeaker is a standard device

with no need to be declared as opposed to the LED, which is an OP device component to be declared and connected to the language component. After testing with both OP prototypes, we found that a LED was less obtrusive than a loudspeaker and chose the LED as an output linking device.

(a)

```
int level;                    // digital property level
int level_old;                // previous value of the property
int angle;                    // digital property level

void setup() {
  pinMode(13, OUTPUT);
}

void feedback() {
  digitalWrite(13, HIGH);  // set de LED on
  delay(1000);             // wait for a second
  digitalWrite(13, LOW);   // set the LED off
}

void loop() {
  int angle = analogRead(A0); // read the value of "angle" from the sensor

  if (level!=level_old) {     // if "level" changed
    feedback();
  }
}
```

(b)

```
  int lowerBound = 42;   // lower bound of the ramp transformation
  int upperBound = 600;  // upper bound of the ramp transformation
  int valueMax = 1023;   // maximum value from the sensor

  if (angle < lowerBound) {
  angle = 0;
  } else if (angle > upperBound) {
  angle = valueMax;
  } else {
  int a = valueMax / (upperBound - lowerBound);
  int b = -lowerBound * a;
  angle = a * angle + b;
}
```

Fig. 12. Prototyping the ORBIS object with Arduino (a) with an identity input linking language and (b) the lines of code to be added for using a ramp transformation.

Through simple examples taken from our design experience in collaboration with a product designer, we illustrated the object level of the OP toolkit and its benefit towards flexibility of the evolving prototypes. Only a few lines of code, easily located, needed to be changed when ideas evolved. Moreover while collaborating with the designer, we clearly observed the usefulness of focusing on both physical appearance and interaction of the object at the same time - such joint activities being possible thanks to the OP toolkit.



```
DigitalProperty level("level", QVariant::Int);
char* soundFile = "";
BeepOutputLanguage beep("beep", soundFile);
QObject::connect(&level, SIGNAL(PropertyUpdated(QVariant)),
&beep, SLOT(update(void)));
isAnalogue = false;
PhidgetInterfaceKitDevice led("led", LinkingComponent::OUT, indexOnBoard,
isAnalogue);
QObject::connect(&beep, SIGNAL(updated(bool)),
                 &led, SLOT(UpdateOutput(bool)));
```

Fig. 13. ORBIS object: Prototyping the output linking modality with a red LED.

```
DigitalProperty level("level", QVariant::Int);
char* soundFile = "./Pop.aiff";
BeepOutputLanguage beep("beep", soundFile);
QObject::connect(&level, SIGNAL(PropertyUpdated(QVariant)), &beep,
SLOT(update(void)));
```



Fig. 14. ORBIS object: Prototyping the output linking modality with a loudspeaker.

The object level also promotes the reusability of objects: we actually reused the designed mixed object in another application context. We easily integrated this turnable object into Google Earth. Fig. 9b shows the prototyped tool in use for rotating the earth. This was the starting point for exploring design solutions of a tool for Google Earth. The mixed object must then be adapted and tuned for this new context of use. This shows the reusability at the mixed object level supported by the toolkit.

5.2 Evaluation of the integrated design approach

Olsen [19] lists 9 possible claims of a user interface system. Three of them aim at *reducing solution viscosity* (reducing the effort to explore many possible design solutions): *flexibility* (rapid changes), *expressive leverage* (accomplish more by expressing less) or *expressive match* (closeness between the means for expressing design choices and the problem being solved). As our claim is the *expressive match* between the toolkit and an interaction model, thus targeting the reduction of *solution viscosity*, we chose to evaluate the toolkit regarding this claim. We therefore applied Olsen's framework for evaluation of *expressive match*:

- Explain the target *situations, tasks and users*,
- State the importance of the problem,
- Then demonstrate the expressive match (i) by measuring time to create a design or express a set of choices, *or* (ii) by challenging subjects to correct a design flaw and by reporting time, errors, difficulties and/or success rates. In this section, we report the second method of evaluation (ii).

The study gathered 4 participants who were developers but novices with the OP toolkit. Pairs of participants were asked to make two different randomly chosen modifications in the code of the ORBIS turnable tool (Fig. 10a). They first carried out the modifications of the output linking device of Fig. 13 and 15, followed by the modifications presented in Fig. 11. Modifications to be done were presented as MIM descriptions (Fig. 10a) without text. Making them work in pairs enabled them to talk more naturally. We provided the documentation of the toolkit for this exercise. We chose to ask them to perform these realistic modifications because we wanted to evaluate if they could figure out how to make a modification that can occur in real design settings. Indeed these modifications come from our prior experience with designers. After completing these two exercises by pairs, we conducted a discussion altogether about their difficulties as well as the identified benefits and drawbacks of the toolkit. During all the experiment, the participants were not given any help or information about the MIM modeling and the toolkit.

We identified one problem: they expected to find a linking device component in OP for the loudspeaker device (exercise 3 - modifications presented in Fig. 13 and 15).

Indeed, it was unnatural for them to change a parameter in the linking language component when asked to change the linking device. This illustrates the relevance of our approach. Even though OP is an improvement towards expressive match between MIM and a prototyping tool, the toolkit should go even further and literally follow the MIM outline, bypassing existing software. The Identity language component is one contribution towards this goal, and we are already working on this identified problem related to standard devices (e.g., screen and loudspeaker).

Encouraging results also came up: During an exercise, participants had to write pseudo-code from a MIM modeling or a text description, and then explain an OP code. One of the participants who was given the MIM modeling of Fig. 2a drew a similar description for explaining the code in the second exercise. This shows that it can be straightforward to go back and forth from the interaction model to an OP code. Moreover after the experiment, one participant said that using OP was even not code writing for him, showing that he found it very easy working with the pair of tools. Finally they all suggested a graphical user interface for the toolkit. We actually plan on providing a tangible interface for OP as a counterbalance to on-screen prototyping.

6 Conclusion

To address the challenge of fluid design of mixed systems, we have presented the OP toolkit for prototyping mixed objects.

OP introduces the object level in prototyping mixed systems, a level not supported by existing toolkits. On the one hand, the OP toolkit is built on top of low-level technological toolkits, but still requires, in its current version, some basic notions of programming for assembling the parts of a mixed object. Its scope includes the scope of Phidgets [22] or BOXES [13] amongst others. On the other hand, the OP toolkit presents a high ceiling by enabling various mixed objects to be connected to applications. We are currently examining the integration of the OP mixed object library into User Interface Management Systems (UIMS) like the one described in [20]. Indeed OP provides support to prototype tools and task objects, so that a UIMS can use them as building blocks for the development of the entire application.

OP provides a set of components for rapidly building functional physical objects that are based on the Mixed Interaction Model [7], a conceptual design model that has been shown to be useful for exploring the design space of mixed objects. The tool allows its users to explore at the same time the physical forms of the object with various materials (foam, play dough, cardboard, etc.) as well as the interaction with the object via seamless conceptual reasoning and practical prototyping.

In the future, we would like to empower new users, namely designers with no programming skill but not necessarily working together with a developer. Toward this aim, we will be exploring a tangible interface for the toolkit. Indeed, prototyping is often participatory and tangible user interfaces suit group work, in contrast to on-screen interfaces. OP could provide tangible blocks for each abstraction level of the mixed object, to be embedded in a rapidly shaped physical prototype. In this way, re-arrangements and iteration directly on the physical prototypes could be done by each member of the group. In addition, a further open challenge we would like to address is in defining a tool based on the OP toolkit for letting the end-users define at runtime a

mixed object by linking physical and digital properties. For example in [5] three types of coupling are defined: [personal, universal, transient] coupling between physical and digital properties of a mixed object that is defined dynamically.

Acknowledgments. We thank N. Mandran (LIG) for her support on evaluations.

References

1. ARTooKit, http://www.hitl.washington.edu/artoolkit/
2. Beaudoin-Lafon, Designing Interaction, not Interfaces, AVI'04, 15-22.
3. Bouchet, et al, ICARE Software Components for Rapidly Developing Multimodal Interfaces, ICMI'04, 251-258.
4. Buxton, Sketching User Experiences: Getting the Design Right and the Right Design, Morgan Kaufmann Publishers Inc. 2007
5. Carvey, et al., Rubber Shark as User Interface, CHI'06, 634-639.
6. Coutrix, Nigay, Balancing Physical and Digital Properties in Mixed Objects, AVI'08, 305-308
7. Coutrix, Nigay. An Integrating Framework for Mixed Systems, In The Engineering of Mixed Reality Systems Book, Springer-Verlag, 9, 2009.
8. Dragicevic, Fekete, Input Device Selection and Interaction Configuration with ICON, IHM-HCI'01, 543-448.
9. Dubois, Gray, A Design-Oriented Information-Flow Refinement of the ASUR Interaction Model, EHCI-HCSE-DSVIS'07.
10. Fishkin, A taxonomy for and analysis of tangible interfaces, Personal Ubiquitous Computing, 8(5), Septembre 2004, 347-358.
11. Hartmann, et al., Reflective Physical Prototyping through Integrated Design, Test, and Analysis, UIST'06, 299-308.
12. Holmquist, et al., Token-based access to digital information, HUC'99, 234-245.
13. Hudson, Mankoff, Rapid Construction of Functioning Physical Interfaces from cardboard, Thumbtack, Tin Foil and Masking tape, UIST'06, 289-298.
14. Jacob et al., Reality-Based Interaction: A Framework for Post-WIMP Interfaces, CHI'08, 201-210.
15. Klemmer, et al., Papier-Mâché: Tookit Support for Tangible Input, CHI'04, 399-406.
16. Lim, et al., The anatomy of prototypes: Prototypes as filters, prototypes as manifestations of design ideas, ACM TOCHI, 15(2).
17. MaxMSP, http://cycling74.com/
18. Myers, et al., Past, Present and Future of User Interface Software Tools, ACM TOCHI, 7(1).
19. Olsen, Evaluating user interface systems research, UIST'07, 251-258.
20. OpenInterface Platform, http://www.oi-project.org
21. Patten, Ishii, Mechanical Constraints as Computational Constraints in Tabletop Tangible Interfaces, CHI'07, 809-818.
22. Phidgets, http://www.phidgets.com/
23. Qt, http://trolltech.com/products/qt
24. Salber, et al., The Context Toolkit: Aiding the development of Context-Enabled Applications, CHI'99, 434-441.
25. Underkoffler, et al., A luminous-tangible workbench for urban planning and design, CHI'99, 386-393.
26. Wakita, Anezaki, Intuino: an authoring tool for supporting the prototyping of organic interfaces, DIS'10, 179-188.
27. Wellner, Interacting with paper on the DigitalDesk, CACM, 36, 7 (July1993), 87-96.