HCIDL: Human-computer interface description language for multi-target, multimodal, plastic user interfaces

Abdelkrim Benamar  
*Abou Bekr Belkaid University, Tlemcen, Algeria*, abk_benamar@yahoo.com

Lamia Gaouar  
*Abou Bekr Belkaid University, Tlemcen, Algeria*, lamia.gaouar@gmail.com

Olivier Le Goaer  
*LIUPPA laboratory – Pau university, France*, olivier.legoaer@univ-pau.fr

Frédérique Biennier  
*INSA, Lyon, France*, Frederique.biennier@liris.cnrs.fr

Follow this and additional works at: https://digitalcommons.aaru.edu.jo/fcij

Part of the Computer Engineering Commons

Recommended Citation

Benamar, Abdelkrim; Gaouar, Lamia; Le Goaer, Olivier; and Biennier, Frédérique (2018) "HCIDL: Human-computer interface description language for multi-target, multimodal, plastic user interfaces," *Future Computing and Informatics Journal*; Vol. 3 : Iss. 1 , Article 9.  
Available at: https://digitalcommons.aaru.edu.jo/fcij/vol3/iss1/9

This Article is brought to you for free and open access by Arab Journals Platform. It has been accepted for inclusion in Future Computing and Informatics Journal by an authorized editor. The journal is hosted on Digital Commons, an Elsevier platform. For more information, please contact rakan@aaru.edu.jo, marah@aaru.edu.jo, u.murad@aaru.edu.jo.
HCIDL: Human-computer interface description language for multi-target, multimodal, plastic user interfaces

Lamia Gaouar, Abdelkrim Benamar, Olivier Le Goaer, Frédérique Biennier

Abstract

From the human-computer interface perspectives, the challenges to be faced are related to the consideration of new, multiple interactions, and the diversity of devices. The large panel of interactions (touching, shaking, voice dictation, positioning …) and the diversification of interaction devices can be seen as a factor of flexibility albeit introducing incidental complexity. Our work is part of the field of user interface description languages. After an analysis of the scientific context of our work, this paper introduces HCIDL, a modelling language staged in a model-driven engineering approach. Among the properties related to human-computer interface, our proposition is intended for modelling multi-target, multimodal, plastic interaction interfaces using user interface description languages. By combining plasticity and multimodality, HCIDL improves usability of user interfaces through adaptive behaviour by providing end-users with an interaction-set adapted to input/output of terminals and, an optimum layout.

Keywords: Model driven engineering; Human-computer interface; User interface description languages; Multimodal applications; Plastic user interfaces

1. Introduction

The interaction with a computer system evolved considerably during the last decades. Technological breakthroughs have profoundly changed the shape of modern computer systems. The desktop computer is no longer the only representative of end user computing. Indeed, the miniaturisation of devices, the expansion of communication networks, and the growing performance of units of computation represent the major changes made to the scope of users. They are now dealing with a wide range of products: mobile phones became true computers; which have succeeded tablets, connected watches, smart TV, etc. whose capacities are increasing a little more each day and are enriched with new features. For these systems to remain usable despite of their complexity, it is necessary to adopt new means of interaction that go beyond the mouse-keyboard-screen triad and stand out from the WIMP interaction paradigm. Thus, a more instinctive representation of information is required, whether input or output of systems. This observation had led to rethink fundamentally the modes of interaction between human and machine. The main purpose is to promote, as much as possible, a natural communication with the end user.

The combination of multiple modalities as input and/or output allows the improvement of both robustness and reliability of the interaction. The work presented by Richard Bolt [1] stands behind the design and development of multimodal applications, where the computer is enriched with new modes
of interaction to support them. In addition, the emergence of mobile computing and the various kinds of sensors that equip mobile devices (accelerometers, gravity sensors, gyroscopes …) enabled the emergence of new modalities [2] such as tilting the phone or changing its orientation. Multimodality significantly reduces the constraints of platforms interaction, such as small screens or uncomfortable keyboards, and limits the recognition errors.

However, innovation in human-computer interaction cannot be limited to the invention of new interaction techniques whose use is combined in the form of multimodal HCI. Due to the current technological context combining diversity of interaction devices and a redefinition of the interaction space, the design and development of HCI now involve new requirements [3,4]. It is not enough to consider conventional HCI, dedicated to a target device in a target location for a determined type of end user. It is right to move from these invariable and unimodal HCI to the multi-target, multimodal and plastic HCI.

Our work led us to study this perspective further. From our investigations, we present HCIDL, our user interface description language. The goal is to improve the usability of user interfaces via adaptive behaviour through an interaction-set adapted to input/output of terminals and an optimum layout. We propose using both plasticity and multimodality as the support for the remodelling of HCI, when the works of literature focus on one of them. Thus, the contributions of our research are as follows:

- We propose a user interface description language inspired by existing modelling languages such as SMUIML [5] and M4L [6] languages, we call it HCIDL. Our modelling language is structured in three packages according to the MVC model. The user interacts directly with the presentation package, which is connected to the interaction control package that determines the modalities of interaction and application services required to be solicited. The Package Model allows to access to data of functional part.
- HCIDL allows support both input and output mobile multimodal interactions. Based on the CARE [7] and TYCOON [8] properties, we consider four cooperations between the modalities: equivalence, concurrency, redundancy and complementarity.
- The abstraction level offered by HCIDL allows existing sensor management on mobile devices.
- To define an optimum layout for application interfaces, we develop two positioning methods. A “linear” approach for simple user interfaces where components line up one after the other. A second “relative” approach, for more complex interfaces, where the position of a component is expressed relative to the position of another component.
- We propose a comparative study based on criteria we defined and encompassing the multimodality, the plasticity, the abstraction level, the heterogeneity and the development approach.

The remainder of this paper is organized as follow: we detail in section 2 the motivations and objectives behind this work. Section 3 sums up the use of multimodality and plasticity in HCI engineering and provide a comparative study. We devote section 4 to the presentation of our HCIDL language, detailing its abstract and concrete syntax, as well as the transformation rules for Android code generation. We conclude this paper in section 5 by exposing the advantages of our approaches and some perspectives.

2. Motivation and purpose

Today more than ever, the information and communication technologies landscape comes in an impressive kind of classes of devices for accessing information. The physical characteristics of these devices (Size of screen, physical or virtual keyboard, sensors …) have prompted us to develop new techniques of interaction constantly in order to improve the usability of these interaction devices, such as touching, shaking, voice dictation, positioning, etc. to name but a few. The variety of fixed/mobile devices and interaction techniques make the task more arduous for HCI developers due to the lack of an appropriate interaction abstraction level, thus requiring multiple versions of the same user interface to be created depending on the physical and hardware variations of devices (e.g fragmentation). Faced with this problem, for more than decade the concept of UIDL, which draws its roots from domains like model-based authoring, seems to be the relevant answer for creating interactive systems as shown by many contributions such as those discussed in the next section.

Among the properties related to HCI, our proposition is intended for modelling multi-target, multimodal, plastic interaction interfaces using UIDLs. The aim of this work is the adaptation of user interface according to the different parameters that, once combined, constitute an interaction device. More precisely:

- Multi-targeting means that the user interface is intended to support multiple interaction devices. The target term in this case refers to fixed or mobile, interaction devices.
- The “plastic” dimension of our approach denotes the aesthetic aspect of user interface adaptation. The dispatching of the user interface components in the display area implies that the interface preserves its usability despite the physical characteristics of the target interaction device.
- The multimodality contributes to the plasticity of user interface. As part of our approach, multimodality offers alternatives to interaction. Whether in input or output, the user interface can adapt both to luxurious interaction devices as well as to interaction devices that are less rich in terms of modalities.

3. Background

To date, many works have adopted a UIDL approach for designing user interfaces. For the most part, research focuses on portability, multimodality, device independence in user interfaces development to name a few. The diversity of
interaction devices existing today (PC, smartphone, tablets, etc.) makes the portability and adaptability of interaction interfaces a recurring research theme in the HCI community. Given the differences between these interaction devices in terms of screen size, interaction resources, etc., it is difficult to ensure an optimal user experience for each configuration. Here we address this problem by combining the use of plasticity and multimodality in a MDE approach. The goal is to make multimodality an actor in the remodelling of the user interface aimed to provide a satisfactory user experience, whatever device configuration for the parameters: display area and interaction resources.

In this section, we propose a brief history of approaches and frameworks which sums up the use of multimodality and plasticity in HCI engineering, and whose contributions represent a turning point in the development of next generation user interfaces. Afterwards, we present the criteria that we used for evaluating our approach. These criteria are built taking into account: the multimodality, the plasticity, the abstraction level, the heterogeneity and the development approach.

3.1. Reference approaches

The concept of multimodal interfaces was introduced by Richard Bolt [1] in his “put-that-there” system, which combines voice commands with pointing techniques (see Fig. 1). Other prototypes were subsequently constructed to test the combined use of direct manipulation and natural textual language [9], or gestures and speech [10]. Later, real-time video processing techniques were used to improve speech recognition by reading lip movements and to control a panoramic image viewing application by combining eye movements and voice commands [11]. According to Laurence Nigay [12], multimodality is the ability of a system to communicate with a user by using different types of communication channels. The use of multiple communication channels allows each channel to compensate for the weaknesses of others, especially when resolving ambiguities, but also allows the redundancy property, in which a task can be accomplished in various ways. This redundancy of inputs is often desirable, especially when the constraints related to the environment are variable. For example, the redundant use of a microphone and manual input via a keyboard allows the user to define his text using speech recognition, or if this feature is missing, to use the keyboard.

Research in multimodality was initially focused on input multimodality (from the user to the system), involving several physical input devices [12–17]. Few studies have been interested in the design of output multimodality (from the system to the user). Nevertheless, we note two theses, Frédéric Vernier [18] and Cyril Rousseau [19], both devoted to multimodal interaction in output. At this stage, tools for the design and implementation of multimodal interfaces are available. Table 1 shows a set of examples of input and output modalities:

Examples of such tools are a Framework for Adaptive Multimodal Environments (FAME) [21], rules-based approaches presented in Ref. [22], or those that offer more comprehensive solutions, such as PetShop [23], but which require strong programming skills in order to encode the mechanisms specific to each agent. We can also note recent approaches where the focus is to provide a way to design multimodal interactions as the work of Hesenius et al. [24]. The authors demonstrate an approach to incorporate multimodal interaction in Gherkin-based user acceptance tests to specify a system’s behaviour, allowing to include different interaction modalities by using the formal gesture description language GeForMT. This approach is strictly directed at human communication and cannot be used to configure applications. Guedes et al. [25] proposes to extend multimedia languages for supporting the development of multimodal user interfaces in the form of a high-level programming framework. The framework integrates user modalities, both user-generated (e.g., speech, gestures) and user-consumed (e.g., audiovisual, haptic), using the NCL (Nested Context Language) declarative language for the specification of interactive multimedia applications.

To study multimodality, it was necessary to define the relationships between the modalities and the way in which the combination can be used by the system. For this purpose, the types of combination of modality were presented more formally by Coutaz and Nigay [7] under the name of CARE properties (Complementarity, Assignment, Redundancy, Equivalence) and extended in Ref. [8] under the name of TYCOON (Types and goals of COOperatioN). An interaction modality in CARE is defined by the pair “d, r” [26] where “d” represents the physical input/output device, and “r” the representational system or the interaction language. Table 2 and Table 3 show some examples of input/output interaction modalities:

Until now, innovation in human-computer interaction was based essentially on the invention of new interaction techniques whose combined use in the form of multimodal interaction interface strove for a more efficient and natural communication between user and machine. But all these techniques assumed the same interaction space: a plain-old desktop computer. However, the diversity and success of handheld computers and mobile phones, the generalization of sensors and networks, the multiplication of embedded systems in everyday objects (cars, television, etc.) have altered the playing field. In this new vision “physical space as a place of interaction”, a user interface called “traditional” quickly proves insufficient. In order to best serve the user and optimize the interactivity capacities of the systems, Thevenin and Coutaz introduce the notion of plasticity of interfaces in Ref. [27] with the first thesis [28] in France on the subject in 2001. The plasticity of an interface indicates its capacity to adapt dynamically to the context of use while respecting its usability [29] or its value [30]. The work of Didon et al. [32] illustrates this definition. The authors propose a software container component based architecture to design adaptive mobile applications that consists of ubiquitous widgets that can dynamically be duplicated, removed and migrated over devices while the application is running. Kalimucho is the proposed software platform to manage their dynamic life-cycle.
MDE is the basis for many, if not the majority, of existing approaches for the development of plastic user interfaces. This fact is not coincidental, but is one of the direct consequences of the characteristics of model-based techniques. Indeed, in MDE the knowledge about the system, the user and its environment are stored into models. The system can thus adapt its behaviour after processing of these models. In 1999, Thevenin and Coutaz [27] propose a conceptual framework, establishing guidelines to promote the development of plastic user interfaces. Based on an MDE approach, the user interface specification is defined by a set of models that combines high level and declarative descriptions of the interaction capabilities of the user interface and the physical environment where it will be executed. These models are then subjected to a series of transformations through automatic or semi-automatic tools until a complete implementation is reached. The work of Thevenin and Coutaz led to the creation of the unified reference framework [32,33] called Cameleon. Cameleon explicates and formalizes various aspects that contribute to the production of plastic user interfaces in terms of models and relationships between models, in a MDE spirit. In addition to the usage context model and adaptation model recommended by Cameleon, the four levels of the Cameleon canvas as shown in Fig. 2, consist of the following models: T&C (Task and concepts models), AUI (Abstract User interface model), CUI (Concrete User Interface Model) and FUI (Final User Interface Model). Although Cameleon is a theoretical framework, it serves as the basis for many approaches in the field of plastic user interfaces [34].

Table 1
Example of input and output sensors [20].

<table>
<thead>
<tr>
<th>Example of input sensors</th>
<th>Example of output sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPS</td>
<td>Loud speaker</td>
</tr>
<tr>
<td>Camera</td>
<td>Vibration</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>Vocal synthesis</td>
</tr>
<tr>
<td>NFC</td>
<td></td>
</tr>
<tr>
<td>Gyroscope</td>
<td></td>
</tr>
<tr>
<td>Compass</td>
<td></td>
</tr>
<tr>
<td>Proximity sensor</td>
<td></td>
</tr>
<tr>
<td>Light sensor</td>
<td></td>
</tr>
</tbody>
</table>

Table 2
Some examples of input interaction modalities [21].

<table>
<thead>
<tr>
<th>Modality</th>
<th>Interaction language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleration</td>
<td>Direct manipulation</td>
</tr>
<tr>
<td>Location</td>
<td>GPS positioning</td>
</tr>
<tr>
<td>Speech</td>
<td>Pseudo natural-language</td>
</tr>
<tr>
<td>Touch screen</td>
<td>Direct manipulation</td>
</tr>
<tr>
<td>Orientation</td>
<td>Direct manipulation</td>
</tr>
</tbody>
</table>

Table 3
Some examples of output interaction modalities [21].

<table>
<thead>
<tr>
<th>Modality</th>
<th>Interaction language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vocal synthesis</td>
<td>Pseudo natural-language</td>
</tr>
<tr>
<td>Widget display</td>
<td>Widgets</td>
</tr>
</tbody>
</table>

Fig. 1. “Put-that-there” system of Richard Bolt [1].

Table 1
Example of input and output sensors [20].

Table 2
Some examples of input interaction modalities [21].

Table 3
Some examples of output interaction modalities [21].
Interaction interfaces description is a particularly dynamic research point and is often based on XML derived languages like UsiXML (USer Interface eXtensible Markup Language) [35]. UsiXML is an UIDL for modelling multimodal mobile-friendly web applications in input and output. It is structured according to the four levels of the Cameleon’s canvas. Modelling multimodal interaction occurs at CUI. The concrete metamodel integrates the modalities of tactile interaction, voice and graphics. The CARE properties of combination between modalities are also processed by UsiXML. UsiXML has a wide range of tools, such IdealXML and GrafiXML [36] for example. They allow the creation of task, abstract and concrete models, the mapping between the models, simulation of task trees and the generation of dialogue models. The final source code of the interface is obtained through generation after many transformations based on these models. The main problem of UsiXML is that the code generation is only web-oriented, when HCIDL allow the generation of native interaction interfaces. Also, when UsiXML doesn’t support mobile sensors-based interactions, HCIDL provide an adequate abstraction level allowing mobile sensors management.

DynaMo-AID (Dynamic Model-Based User Interface Development) [37] is a development framework for context-sensitive user interfaces. It allows modelling of multimodal interfaces in input and output and their cooperation according to the CARE properties. Different models are used to specify interfaces with explicit modelling of the interaction modalities. The task model is made by the engineer as well as the context and presentation models, while the dialogue model is automatically generated by the tool. The models are serialized through a XML-based language called “DynaMOL”. These models are used at runtime to generate the code for the interface in: J2ME for mobile interfaces, Java Swing for desktop applications and HTML/CSS for web applications. The problem with this approach lies in the multitude and diversity of models that can make modelling difficult for a newbie. As mentioned in the next section, HCIDL allows the designer to bring together the interface code in a single file or separate it into several, through a single file extension and a files inclusion mechanism. Similarly to UsiXML, the mobile sensors-based interactions are not taken into account.

SMUIML (Synchronized Multimodal User Interaction Modelling language) [38] is an XML dialect for modelling input multimodal interactions. It was created to configure the platform HephaisTK. SMUIML models multimodal interactions using a dedicated graphical editor as well as their cooperation following the CARE properties. After graphic modelling, SMUIML generates a configuration script used with another Java file to configure HephaisTK attached to the target application. The multimodal dialogue modelling is represented by a state machine in the graphical editor [39]. The editor further offers synchronised dual editing in graphical and textual form as well as a number of operators for the temporal combination of modalities. The main lack in this approach is that the output multimodality is not treated by SMUIML. Moreover, SMUIML/HephaisTK was not created to allow the modelling of interactions with mobile devices. Nevertheless, the language provides an appropriate level of abstraction for model-based mobile sensor interactions (the Recognizer and Trigger concepts in the metamodel). In comparison, HCIDL allows support both input and output mobile multimodal interactions.

Among approaches based on SMUIML, MIMIC [6] is an MDE based approach for development and generation of multimodal mobile applications in input and output, based on SMUIML. The proposed language M4L draws on concepts of SMUIML for the description of events in input and output. Thus, each input/output event is described by: the interaction modality triggered; the cooperation (or not) with other events of the same type on the basis of CARE and TYCOON properties; and the processing triggered by the event. In this approach, cooperation is not considered between modalities but between events that use these modalities for more accuracy when modelling interactions. For designing models, MIMIC uses the meta-editor ModX2 which, from a given metamodel,
generate the corresponding graphic editor. The code generator is implemented as a JavaScript code in this meta-editor. This approach enables code generation for the platforms Android, iPhone and mobile browsers (HTML5/CSS3). Like SMUIML, M4L does not model the exact positions of widgets on application screens and hence cannot generate aesthetic user interfaces.

Hasselt [40,41] is a textual, declarative, event-driven language for the description of executable multimodal interaction models. The core concept of Hasselt is a composite event, which is essentially a user defined sequence of events that are logically related. Within Hasselt, developers define composite events by connecting several primitive events (e.g. touch events or speech inputs) by means of specialised operators. Each operator represents a specific relation between their operands. The overall composite event can then be bound to one or more event handlers, which specify the behaviour the system should expose when the composite event occurs. At runtime, the event handlers are executed every time their associated composite events occur. For event detection, Hasselt relies on existing recognizers to process the low-level input (like speech, mid-air gestures or mouse movements) and does not replace existing recognition-based fusion engines. Hasselt is part of a User Interface Management System (UIMS) suite, called Hasselt UIMS [42]. It includes a code editor, runtime environment and debugging tools for writing, running, and evaluating Hasselt programs.

The focus of these works is limited to the way to design multimodal interactions. In addition to the multimodality, our work covers another angle. We focus on the way to provide an optimum layout for multimodal interaction interfaces. So, we propose two positioning methods that we detail in the next section.

RBUIS (Role-Based User Interface Simplification) [43] is a UI adaptation technique which increase the usability through adaptive behaviour by providing end-users with a minimal feature-set and an optimal layout, based on the context-of-use. The authors define a minimal feature-set as the minimal set required to perform a task, and an optimal layout as one that maximizes the user experience by adapting the properties of concrete UI widgets. RBUIS uses an interpreted runtime model-driven approach based on the Cedar Architecture, and is supported by the Cedar Studio IDE. Thus, user interfaces simplified with RBUIS show a significant improvement in usability compared to their initial counterparts. Closer to our scope, this approach promotes adaptive behaviour of interaction interfaces by providing the minimal feature-set required to a task and an optimal layout. In its definition, HCIDL promotes adaptive behaviour of interaction interfaces through the management of multimodality in order to leverage their interactive potential, while the user evolves in an optimum layout.

Engineering research in HCI has resulted in the emergence of numerous surveys and comparative studies. In his thesis, Bouchet [44] makes a census and classification of known approaches, according to different criteria. These criteria include the coverage of the development cycle, the users targeted (designers, computer scientists, programmers, etc.), the power of expression and the nature of the possible combinations. In their article on multimodal interfaces, Dumas et al. [45] presented a survey of the principles, models and framework in this field. Their study covers issues such as heterogeneous data type fusion, architectures for real-time processing, dialogue management, modelling languages, machine learning for multimodal interaction and frameworks. From this perspective, Dumas et al. [5] published a study on the state of the art of modelling languages of multimodal interaction and enumerated the advantages. The paper highlights a set of guidelines for the design of languages dedicated to the description of multimodal interaction, as well as the roles that this language should focus on: communication, configuration, teaching and modelling. Finally, the authors put forward the balance between particular features of readability and expressivity that these languages must establish.

We have summarized the different stages of plasticity and multimodality to impose them in the field of human-computer interaction in order to arrive at the definition that we know them today. Since then, many methods and frameworks have come to enrich the domain of the new generation user interfaces and contribute to its maturity.

3.2. Evaluation criteria

In order to conduct an objective evaluation of our language compared to the existing works, we have established the following list of criteria based on the recommendations of the literature. We can classify these works according to five characteristics that we consider necessary to respect, so as to ensure a modelling encompassing all the specificities of the development of HCIs (Table 2):

1. **Model-driven UI development approach:** MDE formed the basis for many works researching in UI development. This approach has advantages such as offering technology independence, and providing the ability to support verification of model properties and traceability.

2. **Support for heterogeneity of platforms/devices:** necessary given the rapid evolution of different platforms and the emergence of new interaction devices (PCs, smartphones, tablets ...).

3. **Support for multimodality:** consideration of multimodal interactions in input and/or output. The approach should also integrate modality combinations (CARE properties: Complementarity (C) Assignment (A), redundancy (R) Equivalence (E)).

4. **Support of mobile sensors:** the abstraction level must be high enough to hide the implementation complexity while allowing defining detailed interactions (for tactile pointing modality, it should allow to differentiate between a touch and a long touch, etc.). This will give greater clarity in modelling.

5. **Layout optimization:** optimize layout of user interface by adapting the properties of UI widgets and their positioning to the imposed constraints by interaction devices. To solve
these problems, it is necessary to model the graphic styles with the appropriate level of abstraction.

Table 4 summarize the analysis of these approaches according to the five criteria that we have previously defined. We can see that none of the approaches implements all criteria. This table shows that the current literature brings concrete answers in the management of multimodal interactions using model-based approaches. However, these solutions are not all multi-platform/multi-devices. In addition, we note a weakness of these initiatives in providing sensor-based interaction modalities. But the most relevant observation that we can note is that when approaches focus on designing multimodal interactions, others focus on application design. But none covers both angles. However, we are convinced that the adaptability and usability of interaction interfaces involve leveraging their interactive potential as well as providing an optimum layout. This is what we propose with our approach.

In this article, we complete this range of contributions by presenting a user interface description language named HCIDL. In our approach based on MDE, we combine multimodality with plasticity in a remodelling process. We promote adaptive behaviour of interaction interfaces through the management of multimodality in order to leverage their interactive potential, while the user evolves in an optimum layout:

- Layout optimization: for this we use two positioning methods. A “linear” approach for simple user interfaces where components line up one after the other. A second “relative” approach, for more complex interfaces, where the position of a component is expressed relative to the position of another component;
- Interactive potential leveraging: we use multimodality at this level in order to preserve the comfort of interaction between the user and the machine. The multimodality combined with the CARE properties makes it possible to anticipate the difference in terms of the interaction resources between the different devices.

4. HCIDL: human-computer interface description language

We present in this section the modelling concepts which constitute the abstract syntax of HCIDL (e.g the metamodel) and its associated textual concrete syntax. Subsequently, we present the key rules for model transformations that allow us to obtain the corresponding Android code from the user-defined application model.

![Fig. 3. Global view of the HCIDL metamodel.](image-url)
Fig. 4. Package View of HCIDL metamodel.
We turned to the Eclipse development environment for the implementation of our approach. We rely on the Eclipse Modelling Framework (EMF) to build the metamodel. To implement our textual modelling language, we use the powerful Xtext framework, which from a metamodel generates a corresponding code editor including common useful features like syntax highlighting, auto-completion, code folding, etc. To implement transformation rules for the generation phase, we use the Acceleo framework.\(^3\) The language used by Acceleo is an implementation of the MOFM\(^2\)T standard.\(^4\) This code generation language uses a template approach. With Acceleo, the final source code will be printed out, without any user intervention.

### 4.1. HCIDL abstract syntax

Architecture models define the software organization of the interactive system. They separate the functional part (which implements the concepts specific to the application domain) of the user interface in order to have a better modularity. We propose in this study to apply the principle of modularity to the development of the interface layer of an application. In the spirit of the MVC \(^{[46]}\), our language separates the concerns of developing an application interface into several packages as shown in Fig. 3. It consists of a presentation package (Ihm-View), an interaction control package (Controller) and an interface package to access to data of functional part (Model). The user interacts directly with the presentation package. This package is connected to the interaction controller that determines the modalities of interaction and application services required to be solicited. In the following, we will detail each package of our metamodel.

#### 4.1.1. Package View

As shown in Fig. 4, an interface consists of a screen (metaclass Screen), a navigation bar (metaclass Menu), layouts (metaclass Layout), and a container for graphic components (metaclass GUIElement). The type of structure chosen is specified in the interface model by the boolean meta-attribute isRelative from the class Layout. This feature of our metamodel allows the design of dynamic interfaces that can adapt to any screen size.
Fig. 6a. Package Controller: view on the metaclass InputEvent.
Fig. 6b. Package Controller: view on the metaclass Action.

Fig. 6c. Package Controller: view on the metaclass OutputEvent.
Fig. 7. Package 'Model' of HCIDL metamodel.
Each screen may optionally be composed either of a layout (which themselves may contain other layout or interface components) or interface components (metaclass View-Collection). This is because the metaclass View-Collection is defined as a collection of views (metaclass View). These views can be either layout, either GUIElement. The GUI elements (e.g widgets) are the components that can be found in an interface (button, text field ...). The metaclass StringVA in the above scheme comes from the Model package. It is the link between the Model and View packages.

A special feature of our approach is the management of the visual structure and of positioning of interface components. It is the role of this class. The developer can choose between two kinds of arrangement according to the complexity of the interfaces:

1. Linear: suitable for simple graphical interfaces. Graphics components are simply arranged one after the other in a linear fashion.

2. Relative: the position in the interface of each graphical component is specified relative to the position of another component using the properties provided by the class LayoutProperties. We exemplify them on a simple running example presented in Fig. 5 to facilitate the understanding. In this example, the Ok button is placed under the text field (below=editURL) and its right edge is aligned with the right edge of the text field (align-Right=editURL). Moreover, the Cancel button is placed to the left of the Ok button (toLeftof=Ok) and its upper edge is aligned with that of its neighbour (alignTop=Ok).

### 4.1.2. Package Controller

The package Controller (see Fig. 6) is part of our metamodel responsible for managing the interaction and multimodality. To that purpose, we were inspired by the works of literature such as SMUIML and M4L languages.

The package Controller describes the events and actions related to human–machine interaction. An interaction is represented by the metaclass Control through three points constituting the definition of interaction.

- Fig. 6 a shows the input event represented by metaclass InputEvent. Moreover, the input interaction is modelled as an event issued by the user (clicking a button, scrolling a list, or selecting a menu) or by the system (internal events, as receiving an SMS, or low battery). The class GUIElement in this diagram comes from the package View. It represents the link existing between the two packages Controller and View.

- Fig. 6 b shows the triggered treatment designed by metaclass Action. More precisely, each input event may have effects on the system such as the shift to another application or opening a window.

- Fig. 6 c shows the output event represented by metaclass OutputEvent. Furthermore, the output interaction corresponds to the system response and comes in the form of vibration, voice synthesis, or notification display.

For each input and output event:

- The metaclass InputInteraction/OutputInteraction allows specifying the type (meta-attribute type) and the modality of interaction (meta-attribute modality). This distinction between type and modality of interaction is made as several types of interaction can use the same modality. For example, the interactions long touch and touch both use the tactile modality. This feature of our metamodel allows a very precise description of the interaction which minimizes recognition errors.

- Every event is cooperating or not with other events of the same type (metaclass InputCoop for input events and OutputCoop for output events). Based on the CARE and
TYCOON properties, we consider four cooperations: equivalence, concurrency, redundancy and complementarity. For each event, the developer specifies the nature of the cooperation, the event's name cooperating and the maximum time interval in which must occur both events for the combination to be considered.

4.1.3. Package model

The Package Model describes the resource (data) which can be related to the definition of the interface. As shown in Fig. 7, the resources (metaclass Resource) are defined as the main types of data. For example the metaclass StringResource represents a text resource type. The

```java
1 application "RelativeLayoutDemo"
2
3 screen LoadURL {
4     show layout_LoadURL
5 }
6 Relative layout layout_LoadURL {
7     # label URL {
8         layout:
9             width: wrap_content
10            height: wrap_content
11     }
12     text: "URL:"
13 }
14 # textField editURL {
15     layout:
16         width: fill_parent
17         height: wrap_content
18     toRightOf: URL
19     alignTop: URL
20     marginBottom: 5px
21 }
22 }
23 # button Ok {
24     text: "Ok"
25     layout:
26         width: wrap_content
27         height: wrap_content
28     alignRight: editURL
29 }
30 }
31 # button Cancel {
32     text: "Cancel"
33     layout:
34         width: wrap_content
35         height: wrap_content
36     toLeftOf: Ok
37     alignTop: Ok
38 }
39 }
40 }
```

Fig. 9. RelativeLayoutDemo.lhm.

https://digitalcommons.aaru.edu.jo/fcij/vol3/iss1/9
metaclass `ValueAccess` determines the type of data access: either by literal values (metaclasses ending with “VA”); either by local or external resources declared (class inheriting the metaclass `ResourceAccess`), allowing the variable declaration in the interface model.

Depending on the container-content configuration on which HCIDL is based, the values of the predefined type `LayoutDimensionKind` (see the example presented in Fig. 8) mean:

- `fill-parent`: the component occupies the entire place in the parent container.
- `match_parent`: maximize the size of the component in its parent container.
- `wrap_content`: tells the manager to allocate the minimum size so that the component is rendered correctly.

4.2. HCIDL concrete syntax

HCIDL is a textual modelling language. The description of an application interface requires the modelling of:

- graphical user interface elements and their visual arrangements,
- interactions proposed to allow the user-application communication, and
- data used in the construction of the interface. HCIDL gathers all these points in a high-level language, abstracting away the existing differences between the interaction platforms.

In our description of an application interface, we consider that interface is a set of screens, each describing a state of the application at a given point of time.

Let consider the example of `RelativeLayoutDemo` application which is represented in Fig. 9. It allows entering an URL and to access it. From a graphical point of view, this application consists of a label (lines 7–13) followed by a text field for entering an URL (lines 14–22). Followed by two buttons, `Ok` (lines 23–30) and `Cancel` (lines 31–39). The main screen named `LoadURL` returns to a relative layout (line 4). The attributes `width` and `height` determine the size of elements, such as the value `fill_parent` indicates that the element will occupy all available space in the parent container and value `wrap_content` that it will adapt its size to its content. These properties also have fixed values such as: 5px (pixels), to specify the size of the components.

```
%include <RelativeLayoutDemo.ihm>
when begin
  USER clickOk
  pressed[Ok]
  has (touch, touchScreen)
  equivalence=[{vocal, speech}, 2]
end

do goto "http://www.univ-lemcen.dz"
with begin
  OUTPUT DisplayResult
  has (updateView, display)
end
```

Fig. 10. interaction.ihm.

```
[template public main(alHM : IHM)]
[comment @main]
  [alHM.generateManifestFile()]
  [for (aScreen : Screen | alHM.ihmview.screens)]
    [aScreen.generateScreen()]
    [aScreen.generateLayout()]
  [/for]
  [for (aLayout : Layout | alHM.ihmview.layouts)]
    [aLayout.generateLayout()]
  [/for]
  [alHM.generateResourceFiles()]
[/template]
```

Fig. 11. main.mtl.
In the context of interaction management, the first input interaction is represented in Fig. 10 and implements the click for Ok button (line 4). After entering the URL of the web page, the user can go there in two ways: i) touch-type interaction: by clicking the Ok button and invoking the modality Touchscreen, and ii) vocal-type interaction: by saying the word “OK” through the modality speech. Through the equivalence relation (line 6) existing between the two interaction modalities. Similarly, the second output interaction validates this entry using display modality (lines 9–12) to view the web page (line 8).

The inclusions allow us to include the model files containing some statements. In the previous example (see Fig. 10), it indicates the inclusion (line 1) of the file containing the declaration of the graphic elements of the interface. One of the highlights of our language is to allow the designer to bring together the interface code in a single file or to separate it into several. This modularization character is made possible precisely by keeping the same extension for each file, but this involves managing properly cross-references and scoping. This modularization of the interface code helps to lighten the contents of each file and promotes their reuse.

4.3. HCIDL generation rules

Acceleo allow defining the rules for automatically transforming the model conforming to our metamodel into Android code of the corresponding application. The deployment of the generated application is done using Eclipse. The developer starts by creating an empty Android project under Eclipse, and then gives its address to the generator so that it can generate the java and xml files of the application, following the models of the previous subsection. Our generator consists of a set of template modules. Each module is composed of several templates whose role is to generate a fragment of the code of the Android application. Templates are invoked at the appropriate time to ensure a coherent assembly of the final code.

The generation process starts with the execution of the module main.mtl which is presented in Fig. 11. The root element of our HCIDL language is the IHM class, and therefore

```plaintext
[template public generateManifestFile(aIHM : IHM)]
[file (AndroidManifest.xml, false, 'UTF-8')]
<<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="[aIHM.packageName]" >
...
...
<application
...
<activity
    android:name="[aIHM.himView.screens->first().name]/Activity"
    android:label="@string/app_name" >
    <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    </activity>
[if (aIHM.himView.screens->size()) > 1]
    [for (screen : Screen | aIHM.himView.screens->subOrderedSet(2, aApplication.screens->size()))]
    <activity
        android:name="[screen.name.toUpperCase()]/Activity">
    </activity>
    [/for]
[/if]
</application>
[/manifest]
[/file]
[/template]

Fig. 12. manifest.mtl.
```

https://digitalcommons.aaru.edu.jo/fcij/vol3/iss1/9
the main template must generate all application code from that object. It is from these modules that are invoked the templates of the generator for the creation of the Android application.

4.3.1. Generating android manifest file

The first invocation in previous figure (line 3) allows calling the template responsible for the generation of xml file AndroidManifest. This is the role of the module manifest.mtl which is presented in Fig. 12. Templates are parameterized models of the target platform. The execution of a transformation consists in taking a template and replacing its parameters with the values of the source model. As shown in Fig. 12, the fragment of code allows declaring in the manifest, all activities of the application (lines 19–25) and the screen class is example of an activity in HCIDL.

```java
[template public generateScreen(aScreen : Screen)]
... package [aScreen.eContainer(IHM).packageName]/;
... public class [aScreen.name.toUpperFirst()]/Activity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (aScreen.layout <> null)
            setContentView(R.layout.[aScreen.layout.name]/);
        [if]
        if (aScreen.element <> null)
            setContentView(R.layout.[aScreen.name.toLower()]/_screen);
        [if]
        ...
        [for (action : Action | aScreen.eAllContents(Action))]
        [action.generateCallback()]
        [for]
        }
    [/file]
[/template]
```

Fig. 13. screen.mtl.

```java
[template public generateCallback(action : Action) post (trim())]
// [action.eContainer(View).eClass().name]([action.eContainer(View).name]/) onClick
public void [action.eContainer(View).name]/.[action.eContainingFeature().name]/[View view] {
    [actionActionCode()]
}
[/template]

[template private actionActionCode(action : Action)]
// This is an abstract template and should not be called
[/template]

[template private actionActionCode(action : GoToURL)]
startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse("[action.url]")));
[/template]
...
[/template]
```

Fig. 14. action.mtl.
```java
[template public generateLayout(aLayout : Layout)]
[aLayout.generateLayout(aLayout.name)/]
[/template]

[template public generateLayout(aLayout : Layout, fileName : String)]
[file (res/layout/.concat(fileName.toLowerCase()).concat('.xml'), false, 'UTF-8')]
<?xml version="1.0" encoding="utf-8"?>
generateViewNode(aLayout)/
[/file]
[/template]

[template public generateLayout(aViewCollection : ViewCollection, fileName : String)]
[file (res/layout/.concat(fileName.toLowerCase()).concat('.xml'), false, 'UTF-8')]
<?xml version="1.0" encoding="utf-8"?>
generateLayoutNode(aViewCollection)/
[/file]
[/template]

[template private generateLayoutNode(collection : ViewCollection)]
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    [for (view : View | collection.views)]
        [view.generateViewNode()]
    [/for]
</LinearLayout>
[/template]

Fig. 15. layout.mtl.

[template private generateViewNode(view : View, subViews : ViewCollection) post(trim())]
<view.nodeTagName/>
[if (view.isRootNode())]
    xmlns:android="http://schemas.android.com/apk/res/android"
[endif]
[if (view.name <> null)]
    android:id="@+id/[view.name/"]
[endif]
[view.generateLayoutProperties()]
[if (subViews <> null)]
    [view.generateViewProperties()]
[/if]
[for (view : View | subViews.views)]
    [view.generateViewNode()]
[/for]
</view.nodeTagName/>
[/if]
[/if]
[/template]

Fig. 16. widget.mtl.

https://digitalcommons.aaru.edu.jo/fcij/vol3/iss1/9
4.3.2. Generating application activities

As shown in Fig. 11, the loop in main.mtl (lines 5–8) allows scanning the model and transforming the screens objects to activities of the Android application. This is the role of the screen.mtl module which is presented in Fig. 13.

In the module Action object, a java class is generated for each application activity that is parameterized with the values of the source model. The code needed to receive the input events and perform the associated effects is also generated in the activity to which they apply, from the Action object of the

```
[comment encoding = UTF-8 /]
[module resource('http://www.xtext.org/example/droid/Droid',
  'http://www.eclipse.org/emf/2002/Ecore')]

[template public generateResourceFiles(app : Application)]
  [app.stringResourceFile( )]
  [generateResourceFiles(app.resources->asOrderedSet()->filter(Resource))]
[/template]

[template private generateResourceFiles(resources : OrderedSet(Resource))]
  [stringResourceFile(resources->filter(StringResource))]
  [integerResourceFile(resources->filter(IntegerResource))]
  [booleanResourceFile(resources->filter(BooleanResource))]
  [colorResourceFile(resources->filter(ColorResource))]
  [dimensionResourceFile(resources->filter(DimensionResource))]
  [integerArrayResourceFile(resources->filter(IntegerArrayResource))]
  [stringArrayResourceFile(resources->filter(StringArrayResource))]
[/template]

[template private stringResourceFile(app : Application)]
  [file ('res/values/strings.xml', false, 'UTF-8')]
  <?xml version="1.0" encoding="utf-8"?>
  <resources>
    <string name="app_name">[app.name/]</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
  </resources>
[/file]
[/template]

[template private stringResourceFile(resources : OrderedSet(StringResource))]
  [if (resources->size() > 0)]
    [file ('res/values/gen-strings.xml', false, 'UTF-8')]
    <?xml version="1.0" encoding="utf-8"?>
    <resources>
      [for (resource : StringResource | resources)]
        <string name="[resource.name/]">[resource.value/]</string>
      [/for]
    </resources>
  [/if]
[/template]
```

Fig. 17. resource.mtl.
model. For example, the code fragment of the module `action.mtl` (see Fig. 14) generates the code needed to invoke a web page by its URL.

4.3.3. Generating layouts and interface components

For each activity, a layout is generated for the output events with a display modality. This is the role of the `layout.mtl` module (see Fig. 15). This module is called from line 11 of the `main.mtl` module (see Fig. 11). As shown in Fig. 15, the module `layout.mtl` is composed of two parts; the first one (lines 2–11) is dedicated to the generation of relative layouts while the second one (lines 13–30) is dedicated to the generation of linear layouts.

In its first version, the layout module was responsible for generating both the layout XML files and the widgets XML nodes. In this version we opt for a split of responsibilities. Thus, the `layout.mtl` module is responsible for generating layouts XML files and generation of widgets is delegated to the `widgets.mtl` module.

The fragment of code of the `widget.mtl` module (see Fig. 16) allows creating an xml node for the declaration of each view component.

4.3.4. Generating resource files

Generating application resources is relatively simple. It consists of a set of declarations. This is the role of the `resource.mtl` module which is presented in Fig. 17. For example, the code fragment presented in this figure allows generating in the resource file `strings.xml` of the application, string type resources declared in the model.

5. Conclusion

In this paper, we presented HCIDL, a description language for modelling and generating multi-target, multimodal, plastic user interfaces. Our work is motivated by the difficulty of application interfaces development, combined with the multiplicity and diversity of the existing interaction platforms.

Our approach respects the main criteria of the MDE defining an effective model-based development. Then, we provide a clear and structured textual modelling language. Its originality lies in its structure in three packages according to the MVC model, which accentuates its modularity. The steps of modelling, automatic code generation and application deployment do not require migration and are done in the same environment. At this stage of its development, HCIDL allows supporting both input and output multimodal interactions. The abstraction level offered by HCIDL allows existing sensor management on mobile devices. Through the arrangement and positioning mechanism of interface components implemented in the package “View”, HCIDL defines an optimum layout for application interfaces.

In the short term, we aim at the completeness of our generator in order to be able to target new platforms. Thereafter, we will try, using our metamodel, to identify the most basic combinations of modalities, to model them and to allow their reuse during the modelling of new applications. Afterward, a project for the development of our own UI builder tool based on our HCIDL user interface description language will be considered.

References


