A Gestural Concrete User Interface in MARIA

Lucio Davide Spano
Department of Mathematics and Computer Science, University of Cagliari
Via Ospedale 72, 09124, Cagliari, Italy
davide.spano@unica.it

Fabio Paternò
ISTI-CNR
Via G. Moruzzi 1
fabio.paterno@isti.cnr.it

Gianni Fenu
Department of Mathematics and Computer Science, University of Cagliari
Via Ospedale 72, 09124, Cagliari, Italy
fenu@unica.it

ABSTRACT
In this paper, we describe a solution for engineering and modelling user interfaces for supporting input collected through gesture recognition hardware. We describe how we applied such approach by extending the MARIA UIDL, and how the modelling solution can be applied to other UI toolkits. In addition, we detail the model-to-code transformation for obtaining a running application through an example case study.

Author Keywords
Gestural interaction, Input and Interaction Technologies, Analysis Methods, Software architecture and engineering, User Interface design.

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION
Different gesture recognition devices are available on the market nowadays. They are more and more becoming popular not only in the entertaining field, but also for desktop applications (e.g. the Leap Motion or Microsoft Kinect). Most of the times, the interfaces exploiting such devices are created starting from different widget toolkits that do not support them natively. Developers are therefore required to create bridging code between UI toolkits and the libraries managing the input device. Such separation, while useful for reusing graphic controls also in gestural interfaces, forces developers to redefine functionalities that are usually supported by the underling toolkit, such as the pick correlation.

In this paper, we describe how we extended MARIA [9], a model-based user interface description language, in order to support gestural interaction, taking as starting point the graphical desktop interface meta-model. We identified two problems shared with other UI toolkits: the implicit relationship between the pointing device and the pointer position and the lack of separation between the definition of gestures and their effects on the UI. We propose solutions to these problems, which can also be reused in other toolkits. In addition, we discuss a modelling technique for reusing the definition of platform-independent behaviour, which is useful for other modelling languages that exploit different abstraction levels in their definition.

EICS'14, June 17 - 20 2014, Rome, Italy

http://dx.doi.org/10.1145/2607023.2610282

RELATED WORK
The problem of a more effective integration of the gestures in both the design process and the structure of UI toolkits, has been addressed in [1]. They identified the different stakeholders involved in the design process and a set of extensions points for the currently available UI toolkits (in particular for pen-based interaction) applying an abstraction-level approach for modelling user interfaces, which is similar to the one we propose in this paper. They considered classifier-based gesture recognition, which is affected by a granularity problem: raising a single event when the gesture is completely recognized impedes the definition of intermediate feedback during the gesture performance.

In this paper, we consider a hierarchical modelling technique, which describes the gestures staring from a set of basic building blocks and creates complex ones, allowing intermediate feedback. Different work employ a similar representation with different formalisms like regular expressions [7, 6], Json [4] or Petri-Nets [3]. We adopted the approach described in [11, 12], which provides a description of gestures through the connection of ground terms related to the low-level events raised by the recognition devices, with a set of temporal operators defined through Petri-Nets and providing a set of compositional operators more expressive than regular expressions (see [12]). In this paper, we go beyond the gesture description, providing a solution for integrating it with the other aspects of existing UI toolkits.

Other approaches adopted in commercial UI toolkits, like e.g. the Kinect SDK, include a set of gestural widgets, containing a hard-coded gesture description, without any support for their composition (e.g. two gestures in

sequence). In MARIA, the two aspects (graphic control and gesture definition) are completely decoupled.

BACKGROUND

MARIA [9] (Model-based lAnguage foR Interactive Applications) is a set of XML languages for defining UIs at different levels of abstractions, according to the CAMELEON [2] reference framework structure. The set includes an abstract language that has multiple refinements for the different interaction platform supported.

The Abstract User Interface (AUI) level describes a UI through the interaction semantics, without referring to a particular device capability, modality or implementation technology. In addition, the interface definition contains the behaviour and the description of the data types that are manipulated by the user interface. The data model is defined using the standard XML Schema Definition constructs.

A Concrete User Interface (CUI) in MARIA provides platform-dependent but implementation language independent details of a UI. A platform is a set of software and hardware interaction resources that characterize a given set of devices, such as desktop, mobile, vocal, multimodal etc. From the CUI different final user interfaces (FUI) can be derived and implemented with different technologies (e.g. web-based or standalone).

GESTURAL CONCRETE USER INTERFACE

In this section we describe the MARIA gestural CUI meta-model, providing a refinement of the AUI language and covering the modelling concepts needed by gestural interfaces. Conceptually, the meta-model should describe how the interface appears, how the user can provide input through the gesture-tracking device and how the interface reacts to such inputs. All these aspect are summarized by the following aspects:

1. The description of the interface layout
2. The description of the data provided by the device
3. The description of the gestures and the temporal relationships between them.
4. The description of the effects that the gestures have on the other parts of the interface.

The first point is related to the visual part of the gestural UI. Since MARIA already contains a desktop CUI definition (see [9] for additional details), we extended the graphic controls for supporting the gestural modality. This is a common starting point also for many UI toolkits: the definition of different graphic controls already exists, what is needed is the support for other input devices different from mouse and keyboard.

The second point covers the data received by the recognition device during the gesture performance. The data description needs to be independent from the actual programming language or development toolkit, a requirement for the compliance with the reference framework in [2].

The third point is related to the gesture description. We adopted the solution discussed in [11, 12], which allows describing gestures starting from a set of ground terms representing the device features (e.g. the joint position), which can be connected by means of different composition operators. This separates the gesture description from the UI behaviour, providing the possibility to reuse the gesture definition for different interfaces, and it can be adopted as solution also in other toolkits.

The forth point deals with two different aspects of the gestural UI model. The first one is how to model the visual feedback that the user has to receive during the gesture performance. The second aspect is the need to relate the definition of the UI behaviour at the abstract level and the recognition of the gestures at the concrete one in order to be compliant with the reification concept [2].

Interactors

The first problem for adapting the desktop interactors to the gestural modality is related to their selection mechanism. In all desktop toolkits, the user activates the interactors through a pointing device, whose movements are shown by the screen pointer. The mapping between the mouse position on the physical space and the pointer position on the screen space is not controlled by the UI developer. Instead, in a gestural interface, designers need to define such transformation. Indeed, it is possible to choose different ways for starting the interaction with a concrete UI object. For instance, we can provide a sequential navigation of the different objects represented on the screen through swipe gestures (a left-to-right swipe for selecting the next object and a right-to-left one for selecting the previous one) or the user may directly point with her hand the object to select.

When considering a gestural interaction, the usual selection process can be summarised as follows:

1. When the selection gesture is recognized (e.g. the swipe ends or the user points on object on the screen), the event-handler associated to the gesture recognition calculates which object has been selected through a pick correlation function.
2. After having identified the selected object, the application should provide feedback to the user. This can be done changing a visual property of the selected object (e.g. the border colour).
3. Finally, if the selected object has some behaviour associated to its selection, it must be executed.

All these steps are usually defined in the code activated by the gesture recognition, which is completely written by the application developer, without reusing any toolkit internal procedures such as the pick correlation, the event tunnelling or bubbling.

In MARIA, we extended the definition of the *Interactor Composition* elements (which represent groups of interactors) for easing the definition of such selection pattern: we exposed a property called *focusPoint* for specifying the pointing position. When the coordinates of a focus point are changed at runtime, the composition is responsible for the pick-correlation, either selecting a contained

interactor or forwarding the notification to the nested compositions. With this protocol, the designer is no more in charge of defining the pick correlation between the point and the interactors, but it is possible to model different strategies for the interactor selection with the gestural modality.

It is worth pointing out that the same solution can be applied to UI toolkits and models different from MARIA, extending container elements (e.g. panels, windows etc.) with a focus point property, and to reuse their pick correlation algorithm, which has been already defined for reacting to mouse events.

Modelling device data

The device data modelling depends on the abstraction provided by the specific gesture tracking hardware. For instance, if we consider a multitouch screen, the device data can be modelled with the array of the 2D position of the on- screen touches (usually from 5 to 10).

If the device is able to track the entire skeleton (e.g. MS Kinect), the device data can be modelled with a structure containing the collection of the skeleton joint positions (a 3D point) and the joint orientation (a 3D vector). One instance of this data structure is available for each tracked user. It is possible to model similarly other tracking devices such as the Leap Motion, which provides the position and the orientation of the fingers, together with the orientation of the palm. The remote-based devices can be modelled through their position and orientation in the 3D space. Such data can be referenced in both event handlers and the modelling of the recognition constraints (detailed in the following section).

The changes on the device state are notified following the observer pattern (e.g. multitouch screens) or through streams returning frames at regular intervals (e.g. Kinect). Usually it is not sufficient to consider the current device state for modelling gestures, but we need to calculate differences between the current values and those received at previous notifications or frames. Our data representation contains also a history of the device state during the recognition. The runtime implementation of the model cannot obviously maintain the whole history, but it should maintain only the part that is necessary for the considered gesture model.

Gestures definition

In MARIA, at the AUI level the dialog model already contains elements for expressing the dynamic behaviour of a presentation. It defines the expected sequence (or sequences) of actions that are supported by the interface. We consider the temporal evolution of a gesture as a concrete example of such dialog model. Figure 1 shows the UML class diagram for the gesture model. At the AUI level, the *DialogModel* is associated to a *Presentation* and consists of different *DialogExpressions*. The *GestureExpression* refines such definition at the CUI level, introducing the modelling elements for the gestural interaction. In order to describe the temporal evolution

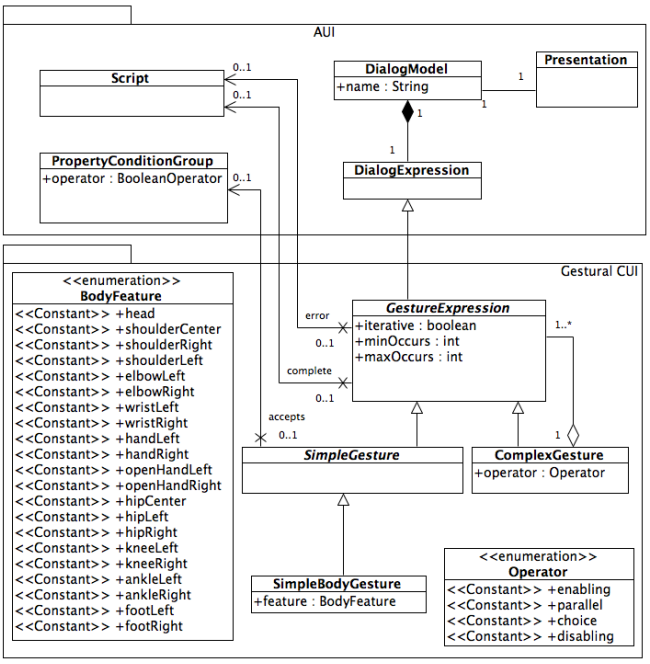

Figure 1. Full-body gesture model

of a gesture, we applied the composite pattern [5]: the gesture description starts from a set of *SimpleGestures* that can be composed in *ComplexGestures*.

A simple gesture recognises a change on a value that is tracked by the recognition device. Figure 1 shows its refinement for the values tracked by a full-body tracking device (e.g. MS Kinect), represented by the *BodyFeature* enumeration. Other devices can be added providing further refinements of the *SimpleGesture* class. In addition, the *SimpleGesture* instances may specify some constraints on such data change for e.g. calculating trajectories. In MARIA this is possible through the instances of the *PropertyConditionGroup* class, which represent Boolean predicates on i) the value of an interactor attribute, ii) the value of a data model element or iii) the result of the execution of an *ExternalFunction*, a functionality that is external to the definition of the UI model, such as a data source or a web service. The latter modelling element allows the reuse of the same predicate across different definitions. *ComplexGestures* are obtained connecting recursively other sub-gestures (either simple or complex) through a set of composition operators, the same used for modelling tasks in CTT [8].

In other UI toolkits it is possible to apply the same modelling solution by organizing the interface definition structure. The code that is responsible for recognizing the gesture must be separated from the code defining the UI behaviour. The simplest way is to use two different classes and to connect them using the observer pattern [5], notifying both the recognition success and error. In addition, there must be a composition mechanism for combining different gesture descriptions (e.g. a composite pattern [5]), specifying different relationships among the composed elements. The description model

may be different from the one we propose in this paper and it may be more or less expressive, but the proposed approach allows isolating this aspect, with the advantage that possible changes in the description model (or even meta-model) would not affect the other UI parts (e.g. the behaviour).

Gesture effects

The hierarchical definition of the gesture model can be exploited in order to attach the UI behaviour to the entire gesture model and/or each one of its subparts, providing feedback not only at the end but also *during* the gesture performance. The UI can react to both the successful and the unsuccessful recognition. In this way, it is possible to define a "rollback" procedure for partially recognized gestures, which restores the UI state. This approach can lead to conflicts between different UI reactions when two different gestures in choice start with the same prefix. This is known as the selection ambiguity problem, and it has been discussed in [12].

In MARIA, the dynamic changes to the UI and to the data model state are defined through the *Script* class. It contains elements representing expressions and statements, which are able to define completely the UI behaviour at the abstract and/or the concrete level (see [9] for more information). In order to distinguish the behaviour associated to the successful recognition from the error management, we connected the generic *Gesture Expression* class with two instances of the *Script* class in Figure 1: the *complete* association defines the reaction to a successful recognition and *error* association defines how to recover a recognition failure.

In MARIA, the interface behaviour defined at the AUI level is inherited by concrete refinements. As we already discussed for the pick correlation problem, the designer may use different paradigms for both the interactor selection and activation. Therefore, the completion of a given gesture should be able not only to trigger the execution of some concrete-platform dependent behaviour, but also to activate the behaviour defined at the abstract level. The binding between the gestures and the abstract events cannot be derived implicitly as in the classical desktop interfaces, but the developer needs to define it explicitly. We rely on raising the abstract events inside the definition of the behaviour associated to a gesture expression in order to solve this problem. The meta-model contains the *Raise* element at the AUI level, which allows raising a specific event (either abstract or concrete) specifying the event name, the interactor identifier and the event arguments (if needed). Therefore, the schema for binding gestures to the abstract behaviour consists of the following steps: i) managing the changes that involve the concrete level (most of the times providing the intermediate feedback) and ii) raising the abstract event that the designer wants to trigger. We provide a modelling example for such binding with a sample application. This solution (including an explicit construct for redefining how the platform-independent behaviour can be activated) can be employed also in other UI toolkits including an AUI level, in order to reuse the behaviour definition for different refinements.

Model to code transformation

A model to code transformation creates the FUI from a CUI definition, exploiting the following technologies:

- WPF as presentation layer
- C# for defining the application behaviour
- The GestIT library for gesture recognition [10, 12]
- The Kinect SDK for managing the sensing device.

The transformation process consists of two steps. The first one transforms the MARIA CUI mode into a XAML file, defining both the UI layout and the gesture description (GestIT provides the XAML tags for the gesture expression). The second step takes as input the same CUI and creates a C# file containing the definition of the application behaviour. Their combination defines the application completely, exploiting the C# partial class definition mechanism. Both transformations are defined using an XSLT, using plain text as output.

It is worth pointing out that the information regarding all the interface aspects (UI appearance and images, gestures and behaviour description) is contained in the CUI model. No other source is exploited for the generation.

SAMPLE APPLICATION

In this section we describe a sample gestural interface modelled in MARIA, a remote controller for a digital TV. The MARIAE tool supports the entire modelling process[1] . The application allows the user to watch a TV show, to change the current TV channel and to retrieve information on the program scheduling. It is organized as follows:

P1 The first presentation contains a video element for watching the TV show. It is connected with P2 through a hidden navigator, which can be activated through a wave gesture (using the greet the screen metaphor typical for Kinect applications).

P2 The second presentation contains two navigators: one pointing to the channel list and the other to the program schedule. The user points one of them with the open hand, and closes it for confirming the selection.

P3 The third presentation shows the channel list in a 3x3 grid. The user can select one element pointing at it and closing the hand or she can change the subset of visualized elements with a hand swipe from left to right (next 9 items) or from right to left (previous 9 items).

P4 The program schedule is shown using a tab container, including an element for each day of the week. The user can go back and forward among the tabs with a swipe gesture.

We start our discussion from the gesture description. In this interface we have three different gestures: wave (P1),

[1]The tool is available at `http://giove.isti.cnr.it/tools/MARIAE/home`

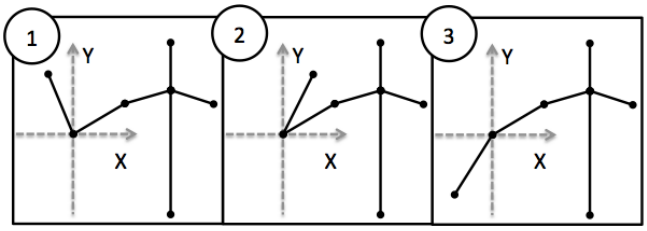

Figure 2. The wave gesture

pointing (P2 and P3) and swipe (P3 and P4). Already in this simple example, the description of two gestures is referenced in more than one presentation. All gestures can be recognized tracking the position of the dominant hand. We assume here to be the right one (the definition for the left is symmetric).

The wave gesture can be defined considering three steps, visually represented in Figure 2. We set the origin of the coordinate system in the users elbow for simplicity in the description: the user starts her hand movement from the second quarter (positive Y and negative X values), then the hand goes in the first quarter (positive X and Y). This sequence must be performed at least once, but it can be repeated more than once. Eventually, the user lowers the hand (third quarter, negative X and Y).

In MARIA, we can model the wave gesture tracking the hand position, as defined in Figure 3. The three steps correspond to the *SimpleBodyGestures*, which differ for the hand positions they accept as valid (contained respectively in the second, first and third quarter of the coordinate system). Such conditions are defined by the accept property of each simple gesture, testing the positive sign of the x and y coordinates of the hand point (posX and posY instances). The conditions for the three steps are respectively $\overline{posX} \wedge posY$, $posX \wedge posY$ and $\overline{posX} \wedge \overline{posY}$. They are modelled by the *PropertyConditionGroup* instances in Figure 3.

The three simple gestures define each step in isolation. In order to define the entire gesture, we need to connect them through the composition operators, defining the *ComplexGesture* instance representing the wave gesture. During each step the hand moves iteratively (the iterative attribute in the *SimpleBodyGesture* instances) until it reaches a position valid for the next step, thus disabling the iteration. The first two steps must be recognized at least once, but they can be repeated an indefinite number of times (respectively the *minOccurs* and iterative attributes of the *cpx1* instance). The gesture is represented by the *wave* instance in Figure 3.

The other two gestures can be defined in a similar way. The pointing gesture consists of an iterative movement of the dominant hand, disabled by the hand closure. The swipe gesture simply consists in a rapid hand movement from left to right or from right to left. This can be modelled through an iterative movement of the dominant hand that maintaining a speed higher than a given threshold (defined in the *accept* property), which is disabled by a

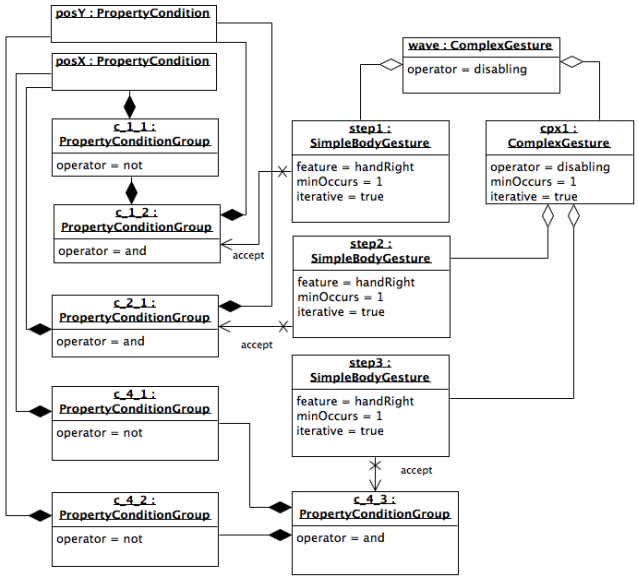

Figure 3. Wave gesture in MARIA

hand movement below that speed. The UI behaviour can be attached to the *complete* and *error* properties of both simple and complex gestures, defining different instances of the *Script* class. In this way, it is possible to reuse the gesture definition in more than one presentation. For instance, the hand closure sub-component of the pointing gesture is associated with the function selection in P2 and with the channel selection in P3, while maintaining the same gesture definition. Since the same gesture can be used in different contexts, designers should include hints on the UI for helping the user in understanding which gestures are available and their effects. We analyse more in detail the definition of P3, in order to show a typical case where redefining the association between the screen pointer position and the physical device is needed. The interface for P3 is shown in Figure 4 (actually it is the result of the model-to-code transformation). The intermediate feedback (a blue border around the channel icon) is associated to the hand movement sub-component of the pointing gesture. The associated *Script* instance is responsible only to project the hand position in the device space on the screen plane (e.g. tracing a line), and to change the *focus point* property of the channel list grouping. The other operations (the pick correlation and the application of the focus styles to the selected element) are delegated to the grouping implementation, as usually happens for e.g. mouse hovering.

The last point we want to detail in this example is the mechanism for connecting the gesture effects and the behaviour inherited from the AUI level. The following actions can be defined at the abstract level in our application, since they are independent from the modality for triggering the interactor (e.g. mouse click, vocal command, gesture etc.) and their definition can be shared on different platforms:

P1 Navigation to P2
P2 Navigation to P3 or P4

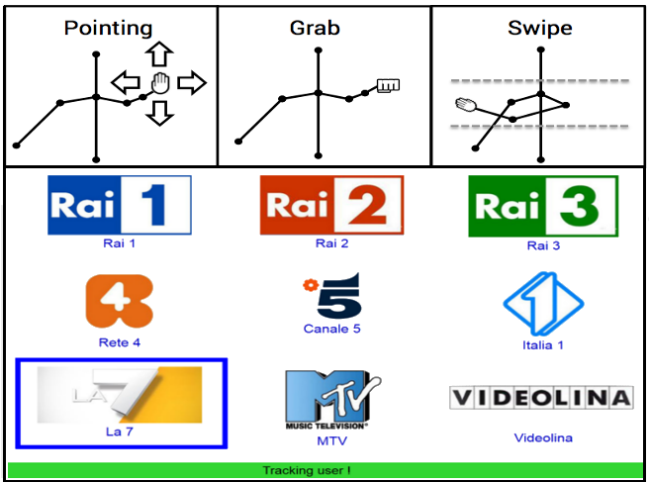

Figure 4. Channel selection interaction and presentation

P3 Changing the video stream (according to the selected channel) and going back to P1
P4 Navigation to P1

Considering the gestural modality, the user does not necessarily point an interactor before activating it: the activation may occur without a spatial correspondence between the gesture and the on screen representation of the interface. For instance, the wave gesture activates the navigation to P2, but the user does not select any link first. Therefore, in order to maintain the conformance to the different abstraction levels [2], we need to connect the completion of the wave gesture with the activation of the navigator between P1 and P2, inherited from the abstract level. In MARIA, this is possible through the *Raise* modelling element, which allows developers to explicitly request the UI runtime support to raise a specific event. In our case, the Script handling the completion of the wave gesture (specified in the complete property) contains a *Raise* element triggering the activation of the navigator between P1 and P2. After that, the UI support executes the event handler associated to the navigator, which has been defined at the abstract level.

CONCLUSION AND FUTURE WORK

n this paper we discussed a new Gestural Concrete User Interface we introduced in MARIA. Through the description of its meta-model, we identified a set of limitations that are common to different UI toolkits and we discussed the solutions we adopted in our modelling language, allowing the separation of four different aspects for defining a gestural UI (interface layout, device data, gesture description, gesture effects). Other modelling languages and toolkits, even with different formalisations of the different aspects, can adopt such organization. In addition, we reported on the model-to-code transformation and detailed the modelling approach through a concrete example.

In future work, we aim to extend the gestural CUI in order to support more interaction devices and to evaluate more in detail the expressiveness of the gesture description model. In addition, we will enhance the tool support with a graphical notation for the gesture description, in order to study the impact of the proposed UI structuring on the design of real-world applications, providing a designer-centred evaluation of our modelling approach.

ACKNOWLEDGEMENTS

We gratefully acknowledge Sardinia Regional Government for the financial support (P.O.R. Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2007-2013 - Axis IV Human Resources, Objective l.3, Line of Activity l.3.1 “Avviso di chiamata per il finanziamento di Assegni di Ricerca”.

REFERENCES

1. Beuvens, F., and Vanderdonckt, J. Designing Graphical User Interfaces Integrating Gestures. In *Proceedings of the 30th ACM International Conference on Design of Communication*, SIGDOC ’12, ACM (New York, NY, USA, 2012), 313–322.
2. Calvary, G., Coutaz, J., Thevenin, D., Bouillon, L., Florins, M., Limbourg, Q., Souchon, N., Vanderdonckt, J., Marucci, L., Paternò, F., and Others. The CAMELEON reference framework. *Deliverable D 1* (2002).
3. Deshayes, R., Mens, T., and Palanque, P. A generic framework for executable gestural interaction models. In *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on* (2013), 35–38.
4. Echtler, F., and Butz, A. GISpL: gestures made easy. In *Proceedings of the Sixth International Conference on Tangible, Embedded and Embodied Interaction*, TEI ’12, ACM (New York, NY, USA, 2012), 233–240.
5. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994.
6. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton++ : A Customizable Declarative Multitouch Framework. In *Proceedings of the 25th annual ACM symposium on User interface software and technology (UIST 2012)*, ACM Press (Berkeley, California, USA, 2012), 477–486.
7. Kin, K., Hartmann, B., DeRose, T., and Agrawala, M. Proton: multitouch gestures as regular expressions. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems (CHI 2012)*, ACM Press (Austin, Texas, USA, 2012), 2885–2894.
8. Paternò, F. *Model-based design and evaluation of interactive applications*. Springer Verlag, 2000.
9. Paternò, F., Santoro, C., and Spano, L. D. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transaction on Computer Human Interaction 16*, 4 (2009), 19:1–19:30.
10. Spano, L. D. Developing Touchless Interfaces with GestIT. In *Ambient Intelligence*, F. Paternò, B. de Ruyter, P. Markopoulos, C. Santoro, E. van Loenen, and K. Luyten, Eds., vol. 7683 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2012, 433–438.
11. Spano, L. D., Cisternino, A., and Paternò, F. A Compositional Model for Gesture Definition. In *Proceedings of the 4th International Conference in Human-Centered Software Engineering (HCSE 2012)*, vol. 7623, LNCS, Springer (Tolouse, France, 2012), 34–52.
12. Spano, L. D., Cisternino, A., Paternò, F., and Fenu, G. Gestit: A declarative and compositional framework for multiplatform gesture definition. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS ’13, ACM (New York, NY, USA, 2013), 187–196.