A Compositional Model for Gesture Definition

Lucio Davide Spano, Antonio Cisternino and Fabio Paternò

ISTI-CNR Via G. Moruzzi 1, 56127 Pisa
{lucio.davide.spano, fabio.paterno}@isti.cnr.it
Università di Pisa, Dipartimento di Informatica, Largo Bruno Potencorvo 3, 56127 Pisa
cisterni@di.unipi.it

Abstract. The description of a gesture requires temporal analysis of values generated by input sensors and does not fit well the observer pattern traditionally used by frameworks to handle user input. The current solution is to embed particular gesture-based interactions, such as pinch-to-zoom, into frameworks by notifying when a whole gesture is detected. This approach suffers from a lack of flexibility unless the programmer performs explicit temporal analysis of raw sensors data. This paper proposes a compositional, declarative meta-model for gestures definition based on Petri Nets. Basic traits are used as building blocks for defining gestures; each one notifies the change of a feature value. A complex gesture is defined by the composition of other sub-gestures using a set of operators. The user interface behaviour can be associated to the recognition of the whole gesture or to any other sub-component, addressing the problem of granularity for the notification events. The meta-model can be instantiated for different gesture recognition supports and its definition has been validated through a proof of concept library. Sample applications have been developed for supporting multitouch gestures on iOS and full body gestures with Microsoft Kinect.

Keywords: Input and Interaction Technologies, Model-based design, Software architecture and engineering, Gestural Interaction.

1 Introduction

In recent years a wide variety of new input devices has changed the way we interact with computers. Nintendo Wii in 2006 has broken the point and click paradigm with the Wiimote controller, based on gestures in a 3D space; iPhone has shown better usability by means of multitouch in 2007, while Microsoft introduced Kinect in 2010 has expressed a way of interaction without wearing sensors of any kind. All these new devices exploit gestures performed in different ways, such as moving a remote, touching a screen, or through whole-body movements.

The introduction of these novel interaction techniques in the mass market has not affected current user interface programming frameworks yet: the underlying model is still bound to the observer pattern, where events occur atomically in time and gets notified through messages or callbacks. Indeed, the support for gestures has been mostly forced in the same paradigm by hiding the gesture recognition logic under the hood, which usually means providing high-level events when the gesture is completed, and leaving the possibility to provide intermediate feedback to the handling of low-level events that are not correlated with the high-level ones. The drawback of this approach is twofold: on the one hand the temporal extension of a gesture is significant with respect to the time scale of a system, a gesture may require in fact seconds to complete; on the other hand the set of recognizable gestures is fixed and inaccessible to the system unless the programmer hooks the low level events generated by sensors and re-implements the full gesture-recognition logic.

Moreover, gesture interfaces exploit the movement evolution in space and time as argument for the interaction, therefore their effect on the UI usually changes according to the movement speed or the space covered, and the program has to perform these calculations. To better handle these problems, a framework must offer an extended vision of how to handle gestures allowing for sub-gesture recognition and concurrent recognition of multiple gestures (i.e. pinch-to-zoom and drawing with another finger). The state of the art tools leave the programmer with the choice between pre-cooked recipes and do-it-yourself with handling low level events. The problem with this paradigm arises when we need intermediate feedback during the gesture execution. With a single event notification, the developer is forced to re-implement the low-level tracking, because it is not possible to separate the complex gesture in smaller constituents. In addition, it is not also possible to compose two or more predefined gestures. For instance, if we want to create a view that can be zoomed and panned at the same time for the iPhone, we have again to track the low-level events, and hard-code the gesture composition.

In this paper, we address such granularity problem by defining a gesture description meta-model that allows constructing complex gestures from a well-defined set of building blocks and composition operators. Moreover, composition semantics is defined using Petri Nets and can be efficiently implemented in a framework. Using our model, developers can associate the UI behaviour either to simple or complex gestures, enabling gestures reuse and composition, which is not currently supported in user interface frameworks. The implementation of a proof-of-concept library for the recognition of the modelled gestures is also described. Finally, we discuss two application prototypes that exploit such library, based on two different recognition supports: iOS devices and Microsoft Kinect.

2 Related Work

The attempt to create compositional representations of different event sources into higher-level events, have a very long research history [1] and the need of a structured approach is exacerbated by the new interaction devices available nowadays. In [2], the analysis of the various interaction techniques considered different dimensions of languages (lexical, syntactical, semantic and pragmatic), in order to provide designers with a theoretical foundation when creating interfaces based on gestures. Through the years, such work evolved in order to include sensors that started to be pervasively included in consumer devices, supporting the advancement towards the *Reality-Based Interaction* [4]. For instance, in [13] the authors characterized the physical actions that users need to perform to enter a command through accelerometer-based devices.

While such categorization is useful to define how and why the user should perform an action rather than another, the following step is to ease the development of such kind of interfaces. In this regard, the support offered by frameworks has to be effective and uniform across the different operating systems and devices [7]. Therefore, we want to focus on generic and machine-understandable gesture descriptions, such as the Gesture Definition Markup Language (GDML) [8], which allows a declarative description of the sequence of events that the device senses for recognizing a custom touch gesture, raising a single event when the gesture completes. We overcome this approach supporting the association of UI behaviour also to gesture sub-parts and parallel gesture recognition. In [6], Kammer et al. described GeForMT, a formalization language for multitouch gestures defined by four components: pose function, atomic gestures, composition operators and the focus on a user interface object. We propose a more general solution that is suitable for devices different from touch screens.

Model-based approaches for user interfaces included gesture descriptions limited to specific supports (e.g. multitouch [11], 3D interaction [3] etc.). In our work, we attempt to overcome the main limitation of these description languages, which is the focus on a single gesture recognition support. Our paradigm aims to be more abstract while allowing concurrent gesture recognition, preserving composition and independence from concrete sensor values: our meta-model allows describing gestures for different recognition devices with a uniform vocabulary.

The definition of a compositional model for complex multitouch gestures has been defined in [5]. The authors use regular expressions for describing gestures, where literals are identified by triples containing the touch event type (start, move, end), the touch identifier and the concerned UI object. The operators are the usual ones for regular expressions. Our work shares with this approach the possibility to create composite expressions for describing gestures, separating them from the UI control. However, we overcome the single gesture recognition assumption in [5], providing the possibility to define parallel gestures. In order to do this, we use Petri Nets for the definition of a gesture meta-model, a notation that has been proved to be effective for the description of event-driven interactive interfaces [1].

In addition, we provide a general solution that is applicable not only for touch devices, but also for different recognition supports by creating an abstract representation of the different features that can be observed, as happens in existing descriptions of multimodal interaction [14].

3 Gesture Description Meta-Model

In this section, we theoretically define our gesture description meta-model. Such meta-model is abstract with respect to a specific gesture recognition support, which means that it is possible to instantiate it for different devices (e.g. multitouch screens, body tracking devices, remotes etc.). We start from the definition of the basic building blocks (ground terms), which represent the set of basic features observable through a specific device. Composed terms represent complex gestures (that can be further decomposed) and they are obtained connecting ground or composed terms through a well-defined set of composition operators.

The definition of the UI behaviour can be associated to the recognition of basic or composed gesture definition. As we will better explain in the following sections, we used Non Autonomous Petri Nets [12] in order to describe the recognition process, since they ease the description of parallel computations driven by external events, such as the reaction of the user interface according to the notifications coming by the gesture recognition device. Once the Petri Nets for a basic building block and for all the composition operators have been defined, the designer can create complex gestures through expressions of basic building blocks and/or complex gestures composed through the set of operators. The actual Petri Net for the complex gesture is derived visiting bottom-up the complex gesture expression definition and can be executed by the library (see section 4.1).

3.1 Basic Building Blocks: Ground Terms

Ground terms of our language are the basic building blocks of our gesture description model, since they cannot be further decomposed. They are defined by the events that developers currently track in order to recognize gestures. Ground terms do not have a temporal extension, though their values may be obtained by computing a function of the raw sensor data (the current gesture support).

For instance, if we are describing a gesture for a multitouch application, the ground terms are represented by the low-level events that are available for tracking the finger positions, which are usually called touch start, touch move and touch end. Besides, for creating full body gestures, the current recognition devices and libraries offer means for tracking specific skeleton points, such as hands, head, shoulders, elbows etc. As happens for multitouch gestures, also full body ones are recognized tracking the skeleton points positions over time.

Here, we define an abstract building block that can be instantiated for different gesture recognition supports. In order to do this, we have to consider that a gesture support provides the possibility to track a set of features that change through the time. As said before, the meaning of each feature (and the associated low-level event) depends on the concrete gesture recognition support.

A feature is a *n*-dimensional vector (e.g., the position of a finger is a vector with two components, the position of a skeleton joint has three components, etc.). A set of features can be also represented with a vector with a number of components equals to the sum of the dimensions of its elements. A set of features is the abstract representation of a gesture recognition support at a given time, since it describes the data provided by a given hardware and software configuration. We will provide examples for the definition of a gesture recognition support in the following sections. The state of a gesture support at a given time is represented by the current value of each feature. The state of a gesture recognition support over time can be represented by a sequence of such states, considering a discrete time sampling. Equation 1 defines a feature f, a gesture recognition support G_S, a gesture recognition support state G_{S_i} and a gesture recognition support state sequence S.

$$
\begin{array}{llll}
f \in \mathbb{R}^n & & & \\
G_S = [f_1, f_2, \dots, f_m] & G_S \in \mathbb{R}^k \;\; f_i \in \; \mathbb{R}^{n_i} & \displaystyle\sum_{i=1}^{m} n_i = k & \\
G_{S_i} = [f_1(t_i), f_2(t_i), \dots, f_m(t_i)] & t_i \in \mathbb{R} & & \\
S = G_{S_1}, G_S, \dots \, , G_{S_n} & n \, \in \mathbb{N} & &
\end{array}
\qquad \textbf{(1)}
$$

A gesture building block notifies a change of a feature value between t_i and t_{i+1}. Such notification can be optionally associated to a condition, which can be exploited for checking properties of the gesture state sequence such as trajectories for hand movements. For instance, it is possible to checks whether the path of a tracked point is linear or not, avoiding the notification of different movements.

The feature change notification is accomplished by the gesture support, and it is external with respect to the current state of the gesture recognition.

We define the basic traits and the composition operators using Non-Autonomous Petri Nets. A Petri Net is a bipartite graph consisting of two types of nodes: transitions (represented as black rectangles) and places (represented as circles), which are connected by directed arcs. A place contains a positive number of tokens and the state of the net is represented by the distribution of the tokens among the places. When all the places that are connected to a given transition contain at least one token, the transition fires, withdrawing a token from all the incoming places and adding one token to all the outcoming ones. In this work, we consider a particular type of Petri Net called Non-Autonomous, in which the firing of a transition is enabled not only by the presence of the tokens, but also by the occurrence of an event that does not depend on the considered Net. Therefore, in Non-Autonomous Petri Net, the transition fires only if the incoming places contain a token *and* if an event of a given type occurs (see [12] for a more detailed description). We need such kind events in order to model the notification of a feature change by the considered gesture support.

We define an event type for each observed feature and each optional gesture state sequence constraint. It is possible to model the external notification with the definition of a function $raise$, which establishes if the Petri Net external event will be raised at a time t as defined in equation 2.

$$
raise\left(E_{f_i,P(S),}t\right) \Leftrightarrow \left(f_i(t) \neq f_i(t-1)\right) \wedge p(S) \qquad p: S \longrightarrow \{true, false\} \qquad \textbf{(2)}
$$

It states that the event $E_{f_i,P(S)}$ at time t is raised if the value of the feature f is changed and if the property on the gesture support state sequence $p(S)$ is verified. In order to avoid a cumbersome representation, we will identify the events simply specifying the related feature and optionally giving a name to the gesture state sequence property. If no constraint has to be verified on the state sequence, we will simply omit it. For instance we will identify the event in Fig. 1 with $f_i, p(S)$. If $p(S)$ is true for all S, the event will be identified as f_i.

In order to model the current progress in the gesture recognition, we use a control state token (Cs) on the Petri Net. The recognition of a basic block will be enabled by the presence of such token, and it will be inhibited by its absence. As we explain better in the following sections, the parallel recognition of different gestures in a composed Net is possible managing multiple instances of such control state token. The Petri Net in Fig. 1 defines a basic building block for gesture recognition.

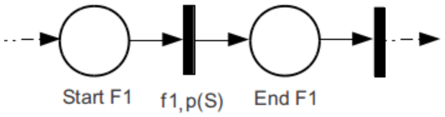

Fig. 1. Gesture recognition building block

The two dotted arrows connect this subnet to transitions that are "externals" with respect to the building block, namely the previous and the following parts of the gesture net. The place $Start\ F1$ receives the control state token from its incoming transition. If it is the first one in the recognition net, it will contain the token associated with the entire recognition process. The transition after this place fires only when the event $f1, p(S)$ occurs. Finally, the control state token will reach the place $End\ F1$, concluding the basic gesture recognition. The actions to be performed in order to react to the basic gesture recognition are associated to the latter place. The out coming arrow starting from the $End\ F1$ place connects the considered block with the next part of the gesture net.

In order to represent a basic building block we use the notation $F_i[p]$: we assign a name to the considered feature (F_i in this case) and also to the boolean function (p), which is omitted if it is true for every gesture support state.

3.2 Composition Operators

A gesture description model is based on the composition of the aforementioned ground terms. The connection is performed through a set of operators, which express different temporal relationships among them. Such set has as starting point those defined in CTT [11], which has been proved effective in defining the temporal relationship for task modelling. Some of them (sequence and choice) have been already defined through Petri Nets in [10]. We provide here a complete definition together with the support for conditions on device feature that is peculiar to gesture modelling. Table 1 lists the composition operators that we will describe in the next sections. All binary operators are associative, therefore the n-ary version of a binary operator (e.g. choice) is defined applying such property.

Table 1. List of composition operators

Operator	Notation	Arity
Iterative	G^*	1
Sequence	$G1 \gg G2$	2 (n)
Parallel	$G1 \mid\mid G2$	2 (n)
Choice	$G1$ [ ] $G2$	2 (n)
Disabling	$G1$ [> $G2$	2 (n)
Order Independence	$G1$ |=| $G2$ |=| ... |=| Gn	n

During the discussion in the following sections, we need also the definition of two different sets of ground terms, given a complex gesture definition. The first one is the set that containing all its ground terms. We refer such set as GS (Ground terms Set). Equation 3 defines how to construct the GS for a gesture G, which consists of a recursive set union on the sub-blocks connected through the composition operators.

$$\begin{aligned}
&G = F_i[p] \Rightarrow GS_G = \{F_i[p]\}\\
&G = G1^* \Rightarrow GS_G = GS_{G1}\\
&G = G1\ op\ G2 \Rightarrow GS_G = GS_{G1} \cup GS_{G2} \qquad op\ \in \{\gg, ||, [], [>\}\\
&G = G1\ |{=}|\ G2|{=}|\ \ldots\ |{=}|\ Gn\ \Rightarrow GS_G = \bigcup_{i=0}^{n} GS_{Gi}
\end{aligned} \tag{3}$$

The second set we need to define contains only the ground terms not appearing as the right operand in a sequencing temporal relation, so they are immediately recognizable when the gesture execution starts. The operators that express such relation are *sequence* and *disabling*. We will call such set starting ground terms set, or SGS and it is defined in Equation 4. Obviously $SGS \subseteq GS$.

$$\begin{aligned}
&G = F_i[p] \Rightarrow SGS_G = \{F_i[p]\}\\
&G = G1^* \Rightarrow SGS_G = SGS_{G1}\\
&G = G1\ op\ G2 \Rightarrow SGS_G = SGS_{G1} \qquad op\ \in \{\gg, [>\}\\
&G = G1\ op\ G2 \Rightarrow SGS_G = SGS_{G1} \cup SGS_{G2} \quad op\ \in \{||, []\}\\
&G = G1\ |{=}|\ G2|{=}|\ \ldots\ |{=}|\ Gn\ \Rightarrow SGS_G = \bigcup_{i=0}^{n} SGS_{Gi}
\end{aligned} \tag{4}$$

3.2.1 Iterative Operator

The iterative operator repeats the recognition of gesture subnet for an indefinite number of times. In order to avoid an infinite gesture definition, each iterative basic block should also be coupled with a disabling operation. As already specified in Table 2, we will use the $*$ symbol in order to represent the iterative operator (e.g. F^* recognizes an infinite number of value changes for the feature one). It is possible to define this operator simply creating a cycle from the ending transition of a gesture subnet to its starting place. In this way, the recognition subnet will be fed again with the control state token, immediately after the gesture has been recognized. Fig. 2 shows the Petri Net definition of the iterative operator. The thicker arrow represents the operator definition.

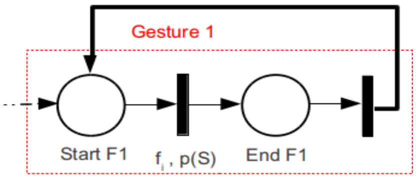

Fig. 2. The iterative operator

3.2.2 Sequence Operator

This operator simply defines that two gesture subnets should be performed in sequence. We use the ≫ symbol in order to represent this operator. It is possible to define such operator connecting the last transition of the first gesture with the starting place of the second one. Fig. 3 shows a gesture consisting of the sequential composition of two basic feature recognizers. The thicker arrow represents the sequence operator.

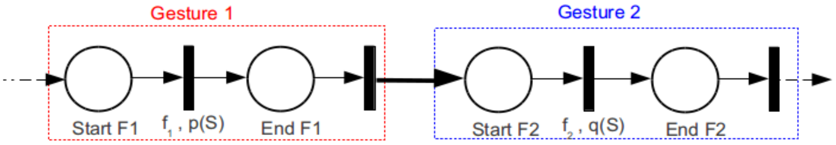

Fig. 3. The sequence operator

3.2.3 Parallel Operator

The parallel operator defines the recognition of two or more different gestures at the same time. We use the || symbol in order to represent the parallel operator.

From the Petri Net definition point of view, the blocks representing the parallel gestures should be simply put in different recognition lines. In order to do this, we assign a different control state token to each line. This can be obtained, as shown in Fig. 4, inserting a transition that "clones" the control state token and dispatching a copy to the starting place of each different recognition lines. We add a place at the end of each recognition line that forwards the "cloned" control state token to the last transition that, once all gestures terminated, restores only one token in the net.

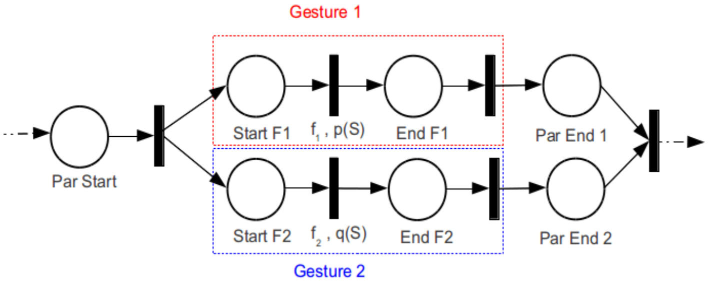

Fig. 4. The parallel operator

3.2.4 Choice Operator

The choice operator defines a gesture that is recognized if exactly one between its first and its second component is detected (either one or the other). We use the symbol [] for representing it. The net can be defined as it is shown in Fig. 5, and its construction is similar to the parallel operator. The transition after the *Choice Start* place splits the control state token between two subnets, each one representing a component involved in the choice. The two lines cannot evolve independently as happens for the parallel operator. Therefore, when one subnet starts its recognition, the other one should be interrupted. In order to do this, it is sufficient to connect the first place of the first gesture subnet with the first transition of the second one and vice versa. In this way, once one of the two feature events is raised, the control state token from the other gesture subnet is deleted.

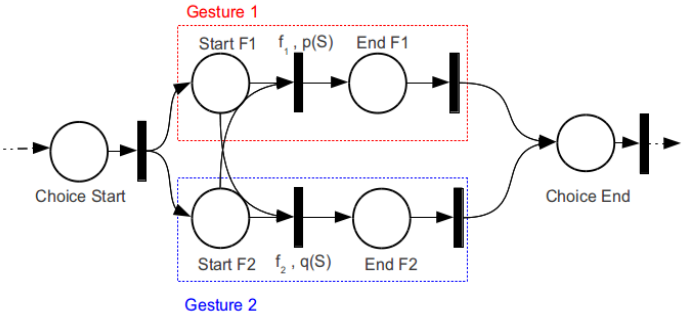

Fig. 5. The choice operator

More precisely the steps to be followed for constructing a Petri Net for $G1[\,]G2$ in the general case are the following:

1. Calculate $SGS_{G1}$and SGS_{G2}
2. Connect the first place of each element of SGS_{G1} with the first transition of each element in SGS_{G2}
3. Connect the first place of each element of SGS_{G2} with the first transition of each element in SGS_{G1}

The last transition of each gesture subnet is connected to the $Choice\ End$ place, which forwards the control state token to the following part of the recognition net.

3.2.5 Disabling

The disabling operator defines a gesture that stops the recognition of another one, thus “disabling” it. The operator symbol is [>. It is typically needed when a gesture is iterative, in order to define the condition that stops the loop. Fig. 6 shows the definition of the disabling operator using Petri Nets for $G1[> G2$. The basic idea is to connect the first place of each basic component belonging to $G1$ to a “copy” of the first transition of the starting blocks of the second one. In Fig. 6 we can see an example of this kind of net, where the first gesture is composed by only one building block. This gesture can be disabled by the second one, which starts with an event related either to the feature $f2$ or $f3$. In order to obtain the desired effect, we connect the $Start\ F1$ place with a copy of both the transitions after the $Start\ F2$ and $Start\ F3$. In order to construct the net for $G1[> G2$ in the general case, we need to perform the following steps:

1. Calculate the sets $GS_{G1}$and SGS_{G2}
2. Connect the starting place of each element of GS_{G1} with a copy of the first transition of each element in SGS_{G2} , possible duplicates (transitions that have the same incoming places and the same external event) are merged. In case of order independence operator, a transition duplicate is added also to each $OI\ Flag$ and $OI\ End$ (see section 3.2.6)
3. Connect the second place of each element in SBS_{G2} with the transitions generated at step 2. Such connection has to preserve the single control state token property for each sub-gesture, so we need to collapse recursively the recognition lines with

an ad-hoc net in the case $G1$ sub-components contain the parallel or the order independence operator.

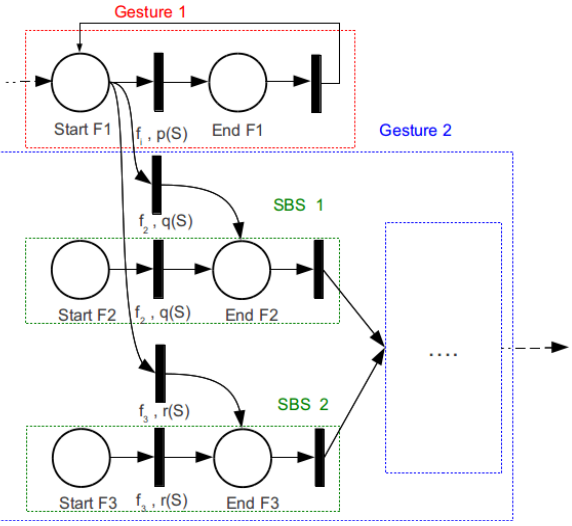

Fig. 6. The disabling operator

3.2.6 Order Independence

The order independence operator is used when two or more gestures can be performed in any order. The composed gesture is recognized when all of its subcomponents have been recognized. We will use the symbol |=| for this operator. It is worth pointing out that such operator is not strictly needed, because it is possible to derive it according to the following property in Equation 5.

$$G1 \; |{=}| \; G2 = (G1 \gg G2)[\,](G2 \gg G1) \tag{5}$$

In general, we can define an order independence composition of a set of n gestures as a choice between all the permutations of its elements. Inside each permutation the gesture set elements are connected through the sequence operator. Obviously, such kind of definition creates $n!$ options for the choice that makes it too expensive both from the space and time point of view. It is possible to provide a more compact net for defining this operator, which is shown in Fig. 7. The idea is to create a Petri Net that repeats n times the choice between the composed subnets, removing one option at each iteration.

The steps to construct this net for $G1|{=}|G2|{=}|\ldots|{=}|Gn$ are the following:

1. Calculate $SGS_{Gi} \forall i \in [1, n]$
2. Create an $OI\ Flag$ place for each Gi and connect it with its last transition.
3. Create an $OI\ End$ place for each Gi and connect it with the same transition at the end of the net.
4. Connect the transition after the $OI\ Start$ place with each starting place of all elements in SGS_{Gi} and with all the $OI\ Flag$ places.

5. For each $i \in [1, n]$, connect the starting places of each element of SGS_{Gi} with all the starting places of each element in $\bigcup_j SBG_{Gj}$, with $\mathrm{j} \in [1, \mathrm{i}-1] \cup [\mathrm{i}+1, \mathrm{n}]$
6. For each $\in [1, n]$, connect the event-driven transitions of each element of GS_{Gi} with $OI\ Flag_i$and vice versa.
7. For each $i \in [1, n]$ connect the ending transition of the net associated to G_i with all the elements in SGS_{Gi}
8. For each $i \in [1, n]$, connect the starting places of each element of SBS_{Gi} with the last transition of the order independence net.

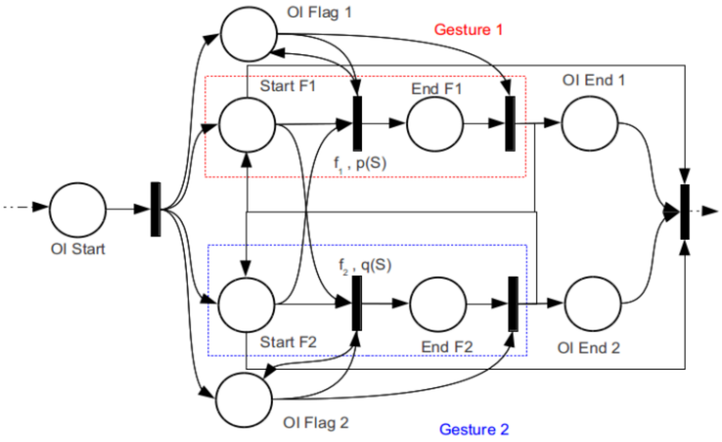

Fig. 7. The order independence operator

4 Library Support

In this section we detail the implementation of GestIT[1], a proof of concept library that allows the development of gesture interfaces according to our meta-model definition. The library class diagram is shown in Fig. 8. It has a core independent from the actual gesture recognition support (*core* package), plus a set of extensions, which deal with the actual devices that currently are iOS devices (*multitouch* package) and Microsoft Kinect (*fullBody* package). The library core contains the classes for the defining gesture expressions (either ground terms or composed ones), represented by the abstract class *TmpExp*. The *SimpleTmpExp* class implements the Petri Net for recognizing a generic basic building block, and it is a refinement of *TmpExp*. The actual feature changes and the optional conditions on them (see section 3.1) are defined by a delegate object associated to the *SimpleTmpExp* instances, which is obviously device-dependent.

Therefore, the library contains an abstract interface (*ExpContent*) that defines the protocol for the generic delegate. It consists of two instance methods: *accept* and *consume*. The first one receives the current gesture recognition support state (represented by the abstract class *ExpEventArgs*) and the Petri Net *Token* that, for

[1] The GestIT library is available at http://gestit.codeplex.com/

convenience, contains the information on the previous gesture recognition support state sequence. A concrete implementation of the delegate returns a boolean value indicating whether the feature change is recognized or not, according to the parameter values. The *consume* method allows the developer to specify the amount of gesture data to be maintained during the gesture recognition, since it is not feasible to maintain the entire sequence of feature values because of memory space. We better detail this point in section 4.1. The possibility to combine building blocks and composed gestures is provided by other two *TmpExp* subclasses: *BinaryTmpExp* and *ComplexTmpExp*. The first one implements all Petri Nets representing binary operators (sequence, parallel, choice, disabling). Obviously, an instance of this class behaves differently according to the *operator* property and its *left* and *right* operands, which belong to the *TmpExp* class (thus it is possible to connect both building blocks and complex gestures). The N-ary versions of such operators can be obtained associating the operands, exploiting the aforementioned associative property. The second *TmpExp* subclass implements the Petri Net for the order independece and contains a list of *operands* (again belonging to the *TmpExp* class). The iterative operator is represented by a boolean flag on the *TmpExp* class.

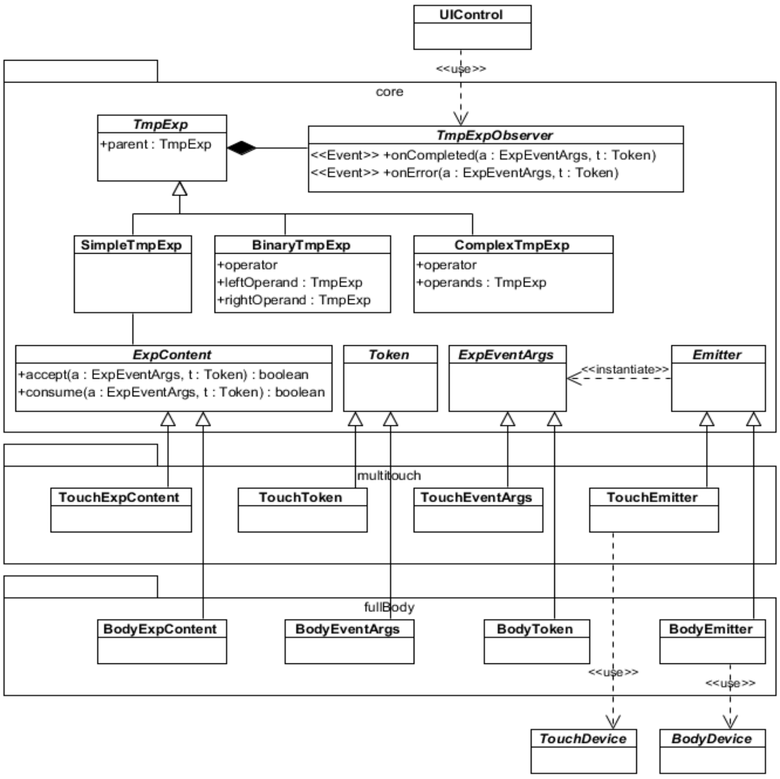

Fig. 8. Gesture Library

A gesture definition is represented by a *TmpExp* tree, where all leafs are *SimpleTmpExp* instances, while the other nodes belong either to the *BinaryTmpExp* or the *ComplexTmpExp* class. At runtime, the tree is managed by a device dependent implementation of the *Emitter* class. Its responsibility is to listen to device updates and to forward them to the leafs that currently contain a token. For each one of them, the *Emitter* will invoke the *accept* method. If the return value is true, the *Emitter* will invoke the *consume* method. Then, the *SimpleTmpExp* will notify the recognition to its parent expression that, according to the Petri Net semantics, will move the *Token*, propagating the notification up to the tree hierarchy and proceeding with the gesture recognition. In section 4.1 we provide a concrete example of this mechanism.

It is possible that the device raises an update that is not *accept*ed by any leaf. In this case, the gesture recognition should be interrupted, and the developer should have the possibility to define how the interface should react to the interruption. The library offers the possibility to associate a handler not only for the successful recognition of a gesture (either basic or composed), but also for the recognition failure. The recognition failure is also propagated to the upper levels of composition tree as in the successful case.

4.1 Modelling Multitouch gestures

A multitouch screen can detect a maximum number of touches. For each touch, the device can detect its screen position (usually expressed in pixel). In addition, it is possible to detect the current time. According to our abstract meta-model, we will have n features related to the touch positions (one for each detectable touch) and a feature related to the current time. If a touch is not currently detected on screen, we say that its current position is the point $(\perp, \perp)$. We identify the feature related to the *i-th* touch with p_i, while we will use the $time$ symbol for the time. In order to have a uniform terminology with the current multitouch toolkits, we define the simplest set of multitouch gestures in Equation 6. Starting from these building blocks, it is possible to define complex gestures using the composition operators. A set of common multitouch gestures is defined in Table 2.

$$\begin{aligned} Start_i &= p_i\,[p_i(t-1) = (\perp,\perp) \wedge p_i\,(t) \neq (\perp,\perp)] \\ Move_i &= p_i\,[\,p_i(t-1) \neq (\perp,\perp) \wedge p_i(t) \neq (\perp,\perp)] \\ End_i &= p_i[\,p_i\,(t-1) \neq (\perp\,,\perp) \wedge p_i\,(t) = (\perp\,,\perp)] \end{aligned} \tag{6}$$

Table 2. Modelling of common multitouch gestures. In expression 2, the condition *pos* checks if the sequence of touches are almost in the same position, while *timeDiff* checks if they are close in time. In expression 5, the path has to be *linear* with a certain *speed*. In expression 6, the *c* condition checks whether the finger movement is circular or not.

	Gesture name	Expression
1	Tap	$Start_1 \gg End_1$
2	Double Tap	$Start_1 \gg End_1 \gg Start_1\,[\,pos \wedge timeDiff\,] \gg End_1$
3	Pan	$Start_1 \gg Move_1^*\,[> End_1$
4	Slide	$Start_1 \gg Move_1^*[linear \wedge speed]\,[> End_1$
5	Pinch	$(\,Start_1 \vert{=}\vert Start_2) \gg ((Move_1^* \vert\vert\, Move_2^*)\,[> (End_1 \vert{=}\vert\, End_2))$
6	Rotate	$(\,Start_1 \vert{=}\vert Start_2) \gg (Move_1^*[c] \vert\vert\, Move_2^*[c])\,[> (End_1 \vert{=}\vert\, End_2)$

In order to recognize multitouch gestures described with this formal definition with our library, we need to define the concrete implementation of the abstract classes discussed in section 4, represented as the *multitouch* package in Fig. 8. The first one is *TouchEventArgs*, an *ExpEventArgs* subclass, which contains the information about a device feature update (touch identifier, touch point, time). The instances of this class are created by a *TouchEmitter*, an *Emitter* subclass, which translates the OS touch screen updates into a format manageable by the library. The *TouchEventArgs* instances are forwarded to the leafs of the *TmpExp* tree that, as already discussed in section4, are *SimpleTmpExp* instances. These leafs are connected with *TouchExpContent* instances, which are *ExpContent* refinements. The *TouchExpContent* class has two instance variables, which represent the touch identifier and the type of a basic building block for touch gestures (start, move, end).

Therefore, the *accept* method checks the conditions defined in Equation 6, according to the specified type. Further conditions to be checked can be defined by developers sub-classing *TouchExpContent* and overriding the *accept* method. The *TouchToken* class contains the information on the gesture sequence, and represents the concrete implementation of a *Token*. Obviously, it is not possible to store in memory each single feature update especially when programming for mobile devices. Therefore, it is possible to specify the maximum number of updates to be buffered and, for convenience, if the starting point of each touch should be maintained or not.

We better clarify how a developer can use the library for providing multitouch gesture support for a UI control with an example. We consider a pinch gesture (defined in Table 2, expression 5) and the following are the steps that have to be followed by the UI control initialization code.

1. Construct the tree of *TmpExps* represented by the UML object diagram in Fig. 9, starting from the leafs, and then associate each *SimpleTmpExp* to the delegate for recognizing the desired feature. This initialization code is generated starting from an XML description of the gesture through an XSLT. However it is possible to code it without the XML description. In addition, it is possible to store such code in a separate class (e.g. *PinchTmpExp*) and reusing it for different UI controls.
2. Create a *TouchToken* instance, specifying the number of updates to be buffered and whether the initial position of each touch has to be stored or not.
3. Create an instance of the *TouchTmpEmitter* class, passing the token created at step 2, and the current UI control (that will be used in order to receive the touchscreen updates from the OS).
4. Attach the handlers to the completion and/or error event of the entire gesture and/or its subparts, represented by the instances of *TmpExp*s created at step 1.

The flow of notifications that allows the library to manage the recognition and to raise the appropriate intermediate events is shown in Fig. 9. We suppose that it has already recognized a touch start with id 1. Therefore, the net is waiting for another touch start, this time with id 2. Such “waiting” is defined by the token position (represented as a circle-enclosed T on the *s2* object in Fig. 9). When the touch screen senses a new touch, the *TouchEmitter* forwards such notification to *s2*, the tree leaf that currently contains the token (arrow 1). After that, *s2* tries to recognize the touch, invoking the *accept* method of its *TouchExpContent* delegate, which will return *true* (arrow 2). Then *s2* notifies its successfully completion to its parent, *c1*, which represents the expression ($Start_1 \mid = \mid Start_2$).

All the building blocks enclosed in this expression are recognized, thus the order independence expression is completed. Therefore, the event handler attached to *c1* is executed. In our example, it paints two circles on the currently visualized image in correspondence of the touch points (A square in Fig. 9), providing intermediate feedback to the user while executing the gesture.

This is the point where our approach break the standard observer pattern: the gesture recognition is not already finished, but it is possible to define UI reactions to the completion of its sub-parts, without re-coding the entire recognition process, as happens for instance when a viewer has a built-in pinch for zoom gesture recognition. After that, *c1* notifies the completion to its parent, *pinch* (arrow 4), which represents an enabling expression. Having completed its left operand, *pinch* passes the token to its right operand *b2* (arrow 5), which represents a disabling expression, and *b2* passes the token to both its operands (arrow 6), which both duplicate it (arrow 7) at next step. The left one represents a parallel expression, while the right one represents an order independence (see section 3.2.3 and 3.2.5). Finally, we have four different basic gestures that can be recognized as next ones: touch 1 move, touch 2 move, touch 1 end or touch 2 end. The dotted circles in Fig. 9 represent the new token positions.

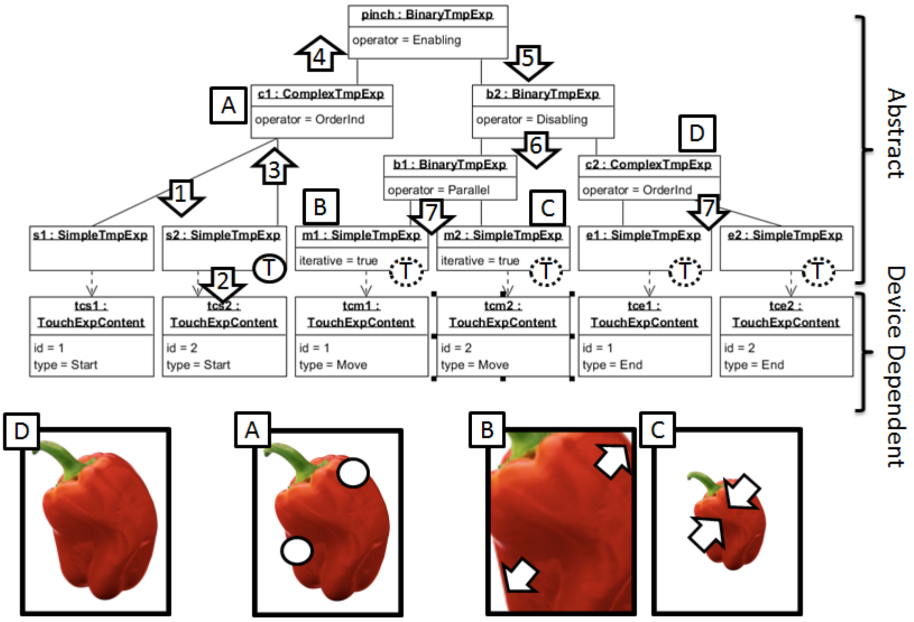

Fig. 9. Recognition of a pinch gesture. The numbered arrows represent the sequence of notifications when the user touches the screen with the second finger, the squares represent the handlers attached to gesture sub-components, while the circle represents the position of the token before the second touch, and the dotted circles the position of the token after the second touch. The lower part shows the effects of the attached handlers on the UI.

It is worth pointing out that the device dependent part of the recognition support is concentrated on delegates for the *SimpleTmpExp* object (represented at the bottom of the tree in Fig. 9). Therefore, the remaining part of the support is implemented by classes that are not bound to a specific device (identified by the “Abstract” label in Fig. 9) and can be exploited not only for multitouch, but also for full body gestures and other recognition supports. The example discussed here is a part of an iOS proof of concept application that allows zooming the current view through the pinch gesture and drawing with a pan gesture. The application gives intermediate feedback during the pinch, showing two divergent arrows while zooming in and two convergent arrows while zooming out (respectively square B and C in Fig. 9). The two gestures are composed through the parallel operator, so it is possible to draw and to zoom the view in at the same time (e.g. using one hand for zooming and one for drawing). From the developer point of view, the difference in handling them at the same time or separately is a matter of selecting the choice or the parallel operator for the composition. No further code is required, which is not the case for current multitouch frameworks. In addition, both gestures have been defined separately from the application (they are contained as samples in the iOS library implementation) and nevertheless the developer can associate UI reactions at different levels of granularity (to the whole gesture, or part of it).

4.2 Modelling Full-Body gestures

The devices that enable the recognition of full-body gestures (e.g. Microsoft Kinect), are able to sense the 3D position of the complete skeleton joints for up to two users, while they can sense the body centre position of up to four more users, in meters. The SDKs provide facilities for projecting the position on the image space of the RGB camera or depth sensor, obtaining the corresponding coordinates in pixels (obviously, without considering the depth axis). In addition, some of them (e.g. Primesense NITE) are also able to track the joint orientations. Finally, it is also possible to have more information using Computer Vision techniques. For instance, it is possible to detect fingertips if the user is really close to the sensor, or to detect if a hand is open or not at intermediate distances (e.g. calculating the convex hull and convexity defects).

From the point of view of our abstract meta-model, we consider as a feature the user identifier, the 3D position of each skeleton joint (both in meters and in pixels), the orientation of each joint (represented as 3D vectors) the time and, if present, any additional information on the hands state (either fingertip position or a hand open or closed flag). As should be clear from the discussion in sections 4 and 4.1, it is possible to extend the library with an *Emitter* subclass (*BodyEmitter*) and a set of *ExpContent* subclasses for recognizing each feature (see Fig. 8, *fullBody* package).

We implemented the library extension in C# with the Kinect for Windows SDK, together with a sample application based on it. The application visualizes a 3D car model, which can be moved and rotated by the user. In order to avoid unwanted interactions, we specified that the user has to stand with the shoulders in a plane (almost) parallel to the sensor, before starting the interaction with the car. Thus, if the user is not in front of the device (which means most of the times in front of the screen), the interface will not give any response. The car position can be changed with a “on air grab” gesture (closing the right hand, moving and reopening it). In addition, the car can be rotated performing the on air grab gesture with two hands, which means closing two hands, moving them maintaining almost the same distance in between, and then reopening them. We want also to display the 2D projected hand position on the screen, in order to provide an immediate feedback to the user for each hand movement. The resulting gesture model is defined Equation 7. The *Front* and *NotFront* gestures respectively activate and deactivate the UI interaction. When a change in the feature associated to the left and right shoulder (indicated as S_l and S_r) occurs, they respectively check if the sensor parallel plane property (p) is true or false.

The UI interaction consists of three gestures in parallel. The first and the second one are simply a hand position change. The UI will react to their completion moving a correspondent (left or right) hand icon. The *Move* gesture is the one associated to the car position change, and consists of a sequence of a right hand close (represented cH_r) and a unbounded number of right hand moves (mH_r^*), interrupted by the opening of the right hand (oH_r). The *Rotate* gesture is represented by the same sequence, performed with both hands in parallel, almost maintaining the same distance (the d condition).

$$\begin{aligned} & Front \gg (mH_r^* \,||\, mH_l^* \,||\, \;(Move\;[\,]\;Rotate))^*[> NotFront \\ & Front = \;(S_l[p]||S_r[p]) \\ & NotFront = \;(S_l[!\,p]||S_r[!\,p]) \\ & Move = \;cH_r \gg (mH_r^*\;[>\;oH_r) \\ & Rotate = \;(cH_r||cH_l) \gg ((mH_r[d]||mH_l[d])^*[> (oH_r||oH_l)) \end{aligned} \tag{7}$$

The intermediate feedback associated to different sub-parts of the composed gestures is shown in Fig. 10. When the correct pose is detected (the *Front* gesture is completed), the car passes from a grayscale to a full-colour visualization, indicating that it is possible to start the interaction (the B square in Fig. 10). When the user “grabs” the car with one hand (completes cH_r), a 4 arrow icon is shown on top of the car (C square). The change of the car position is associated to the following hand movements (mH_r^*). When the user closes two hands in parallel (completes $(cH_r||cH_l)$), a circular arrow is displayed (D square), suggesting the gesture function. The car rotation is associated to the parallel movement of the two hands (the completion of $(mH_r[d]||mH_l[d])\,^*$). The car returns inactive when the user is not in the front position any more (A square).

Writing such application with the support of our library has a set of advantages, which is possible to notice also in this simple case. First of all, the defined gestures are separated from the UI control. Indeed, the car viewer is a standard WPF 3D viewport, enhanced with full body gestures at the application window level. Second, the possibility to inspect the gesture definition and to attach handlers at the desired level of granularity allowed us to define easily when and how to react to the user input, without mixing the logic of the reactions with the conditions that need to be satisfied for executing them. Finally, we do not define any additional UI state for maintaining the gesture execution. Indeed, if we created such application simply with the Kinect for Windows SDK, we would have needed at least a state variable for maintaining what the user has already done and, consequently, for deciding what s/he is allowed to do next (e.g. when the user closes the right hand the state has to change for moving the car at next hand movement). Most of the times, this ends with the implementation of a state machine inside the handler of the skeleton tracking update, which mixes the management of all gestures together. Especially when we want to support parallel gestures, mixing the different gestures leads to code difficult to understand and maintain. Our approach helps the developer to separate the temporal aspect and the UI reaction and to reuse gesture definition in different applications, while maintaining the possibility to define fine-grained feedback.

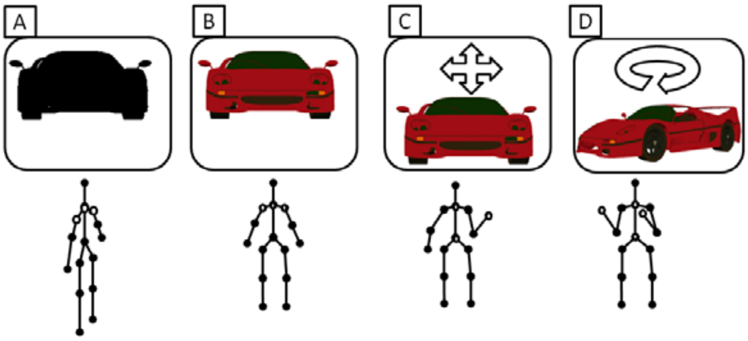

Fig. 10. The car viewer application. The upper part of the figure shows the UI feedback provided while performing the gestures represented in the lower part.

5 Conclusions and Future Work

The lack of proper programming models for defining gestures is a major issue in defining gesture-based interfaces and it limits significantly the ability to fully exploit the new multitouch and 3D input devices, now becoming widely available. The observer pattern underlying the traditional event-based programming is largely inadequate for tracking gestures made of multiple inputs over time, forcing the programmer to choose between handling the complexity of this process or picking one of a pre-defined gestures recognized by the framework used.

In this paper we have proposed a declarative, compositional meta- model for defining gestures, addressing this key issue allowing for simultaneous recognition of multiple gestures and sub-gestures under control of the programmer rather than the framework. The meta-model elements contain ground terms and composition operators that have been theoretically defined using Non Autonomous Petri Nets. It allows reusing and composing the definition of gestures in different applications, providing the possibility to define UI reactions for the recognition not only for the entire gesture, but also for its sub-components. Moreover, we reported a proof-of-concept library, which has been exploited for managing two different gesture recognition supports (iOS and Microsoft Kinect), showing the flexibility and the generality of the approach. We developed two sample applications for demonstrating the advantages of the proposed modelling technique in reusing gesture definitions, which can be exploited at the desired level of granularity.

Now that we have a well-define model we will continue our research by studying both implementation efficiency and effectiveness in real world scenarios. Moreover we will use the ability of the model to recognize many gestures at once to study what we call *posturing*, which is the analysis of user postures while interacting with a system in order to adapt the interface without explicit commands.

In addition, we also plan to provide an authoring environment for the gesture definition, providing testing and simulation capabilities, in order to ease the development of gestural interfaces based on our model and library.

Acknowledgements. This work has been partially supported by the SERENOA project, http://www.serenoa-fp7.eu/.

6 References

1 Accot, J., Chatty, S., Palanque, P. A formal description of low level interaction and its application to multimodal interactive system 3rd DSVIS EUROGRAPHICS, pp. 92-105, Springer Verlag, 1996

2 Bastide, R. Palanque P. A., A Petri-Net based Environment for the Design of Event-driven Interfaces, In *Proc. of 16th International Conference on Application and Theory of Petri-Nets*, (Torino, Italy, June 1995), Springer.

3 Buxton, W. Lexical and pragmatic considerations of input structures. *SIGGRAPH Comput. Graph.*, 17(1):31-37, 1983.

4 González-Calleros, J.M., Vanderdonckt, J., 3D User Interfaces for Information Systems based on UsiXML. In *Proc. of 1st Int. Workshop on User Interface Extensible Markup Language UsiXML'2010* (Berlin, Germany, June 2010), Thales Research and Technology France, Paris, 2010

5 Jacob, R.J.K., Giroaud, A., Hirshfield, L.M., Horn, M.S., Shaer O., Solovey, E.T., Zingelbaum, J. Reality-based interaction: a framework for post-WIMP interafaces. In *CHI 2008*, (Florence, Italy, April 2008), ACM Press, 201-210

6 Kin, K., Hartmann B., DeRose. T., and Agrawala, M., Proton: multitouch gestures as regular expressions. *In CHI 2012* (Austin, Texas, U.S. May 2012)., 2885-2894

7 Kammer, D., Wojdziak, J., Keck, M., Groh, R., Taranko, S. Towards a formalization of multi-touch gestures. In *ITS '10, ACM International Conference on Interactive Tabletops and Surfaces* (Saabrucken, Germany, November 2010), ACM Press, 49-58.

8 Luyten, K., Vanacken, D., Weiss, M., Borchers, J., Izadi, S., Wigdor, D. Engineering patterns for multi-touch interfaces. In *EICS 2010, Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems.* (Berlin, Germany, June 2010), ACM Press, 365-366

9 NUI Group, Gesture Recognition., http://wiki.nuigroup.com/Gesture_Recognition, Website retrieved: 2012-05-27

10 Palanque, P., Bastide, R., Sengès, V.: Validating interactive system design through the verification of formal task and system models. In EHCI 1995, (Yellowstone Park, USA, 1995), pp. 189-212, Chapman & Hall.

11 Paternò, F. Model-based design and evaluation of interactive applications. *Applied Computing*, Springer 2000

12 Paternò, F. MARIA Paternò, F., Santoro, C., Spano, L.D.: MARIA: A Universal Language for Service- Oriented Applications in Ubiquitous Environment. *ACM Transactions on Computer- Human Interaction* 16(4), 1–30 (2009)

13 René, D. Alla, H. Discrete, Continuous and Hybrid Petri Nets, Springer 2005.

14 Scottidi, A, Blanch, R., Coutaz, J. A Novel Taxonomy for Gestural Interaction techniques based on accelerometers. In *IUI '11, Proceedings of the 16th international conference on intelligent user interfaces*, (Palo Alto, CA, USA, February 2011), ACM Press, 63-72.

15 Vanacken, D., Boeck, J.D., Raymaekers, C., Coninx, K.: NIMMIT: A notation for modeling multimodal interaction techniques. ;In *GRAPP 2006*, (Setubal, Portugal, 2006), 224-231