PetriNect – A Generic Framework for Executable Modeling of Gestural Interactions

Romuald Deshayes[1], Philippe Palanque[2], Tom Mens[1]

[1] Service de Génie Logiciel, Institut COMPLEXYS, Université de Mons, Place du Parc 20, 7000 Mons, Belgium
[2] ICS Research Team, IRIT, Université Paul Sabatier, 31062 Toulouse, France

{romuald.deshayes, tom.mens}@umons.ac.be palanque@irit.fr

Abstract. Integrating new input devices and their associated interaction techniques into interactive applications has always been challenging and time-consuming, due to the learning curve and technical complexity involved. Modeling devices, interactions and applications helps reducing the accidental complexity. Visual modeling languages can hide an important part of the technical aspects involved in the development process, thus allowing a faster and less error-prone development process. Even with the help of modeling, a gap remains to be bridged in order to go from models to the actual implementation of the interactive application. In this paper we use ICO, a visual formalism based on high-level Petri nets, to develop a generic layered framework for specifying executable models of interaction using gestural input devices. By way of the CASE tool PETSHOP we demonstrate the framework's feasibility to handle the Kinect and gesture-based interaction techniques. We validate the approach through two case studies that illustrate how to use executable, reusable and extensible ICO models to develop gesture-based gaming applications.

Keywords: model-driven engineering – gestural interfaces – executable modeling – Petri nets – unconstrained interaction

1 Introduction

The last decade has been very fruitful for the development and acceptance of novel forms of interaction. Gestural unconstrained interaction has gained widespread use since the release of new input devices such as Kinect, wiiMote and multi-touch sensing surfaces. Gestural interaction was proposed many years ago as in the multimodal system proposed by Bolt [3] but it took nearly 30 years to be widely available in the mass market. With the release of Kinect[1], a consumer 3D sensor packed with powerful real-time algorithms to track a user's body without using any handheld controller, numerous applications demonstrating the capabilities of the 3D sensor were published on the internet [2,3,15].

[1] www.xbox.com/KINECT

The technical complexity involved in the development of reactive interactive applications is very high, because developers need to address all low-level aspects regarding input and output devices in order to be able to tune adequately the interactions. Beyond that, the toolkits and programming environments proposed by manufacturers integrate such innovations only after they have raised enough interest to developers and provide too little support.

The usual way to fight this complexity is by raising the level of abstraction, separating concerns, and providing generic and reusable software architectures. It is for this reason that we propose PETRINECT, a generic and reusable framework for gestural interaction. The framework is built using ICO, a powerful visual and executable model-based approach relying on high-level Petri nets [11]. Our framework enables the specification of executable models that receive gestural input from devices such as the Kinect. The raw Kinect data is converted into abstract events that are processed by the models and interpreted by virtual objects on an output device such as a graphical user interface or 3D rendering engine. Our gestural interaction framework has been realized using PETSHOP, a tool dedicated to the specification and execution of interactive systems based on ICO models. As a proof of concept, we develop two different case studies to illustrate the expressiveness and reusability of our framework. This contribution counters the critique of [17] that Model-based approaches "model the previous generation of user interfaces".

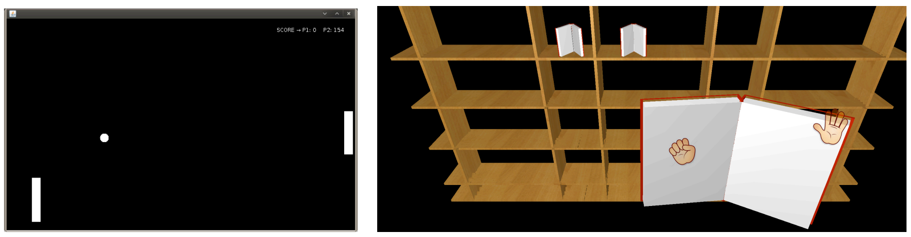
The remainder of this article is structured as follows. Section 2 presents the two case studies and gives some insight about the requirements to create a generic executable modeling framework for gestural interaction. Section 3 introduces the tool and modeling language we have used. Section 4 explains the architecture of our layered framework. Section 5 discusses the lessons learned from these case studies and presents related work. Finally, section 6 concludes.

2 Case Studies and Generic Gestures

The first case study is a Pong game, based on the eponymous two-player game that was very popular in the 70's[2]. The goal of the original game was to prevent the opponent (the computer or a human player) from returning the ball with his paddle, just like in regular table tennis game. The original game was in 2D and the ball could bounce on the upper and lower parts of the gaming area. Our version of the game, developed using the Java Swing UI (Fig. **1**(a)), uses a gestural interface to be able to play the game by using hand gestures instead of a joystick. The left hand controls the position of the paddle on the screen. As the player moves his hand, the paddle follows, thus allowing a natural and simple interaction. We also added the possibility to grab the ball by closing the hand when the ball is close to the paddle. When the player reopens his hand, the ball is released and the game continues. To recognize and track hand movements, we used Kinect's 3D sensor to track a person's body in real-time.

[2] www.pong-story.com

The second case study is considerably more complex. The user can interact with a virtual bookshelf on the screen using hand gestures (Fig. **1**(b)). By moving an open hand, the user can browse books stored on the shelf. He can also drag a selected book towards him. To open the book he has to move both closed hands away from each other. Then, a drag from right to left (or left to right) will turn a page from the book. Finally, approaching both closed hands will close the book, and a drag towards the screen will place the book back on the bookshelf. A YouTube video of this case study is available here http://youtu.be/m9NIvZpQyjs



(a) The Pong game (b) The bookshelf application

Fig. 1. Screenshot of the two case studies**.**

As apparent from the case studies, gestural interaction requires different combinations of hand gestures. Therefore, we provide a *generic* set of gestures that may be reused in various applications to interact with a wide variety of different virtual objects. We extracted the following generic hand gestures from the case studies:

- **Close/Open:** Close (respectively, open) a hand; a parameter indicates which hand.
- **Move:** Move an open hand; a parameter indicates the hand and its direction.
- **Drag:** Move a closed hand; a parameter indicates the hand and its direction.
- **ExpandClose:** Drag both hands away from each other.
- **ShrinkClose:** Drag both hands towards each other.
- **ColinearDrag:** Drag both hands in different directions while being colinear.
- **NonColinearDrag:** Drag both hands in different directions while being non-colinear; a parameter indicates the direction of both hands.

We have given a stereotype name to each different gesture (indicated in bold in the list above). These gestures are generic as they have no concrete behavior associated to them, and can therefore be applied in different ways to different types of virtual objects. The above list of gestures is deliberately incomplete, as it is impossible to enumerate all possible gestural interactions using two hands. To cope with this issue, our goal is to create an extensible framework for gestural interaction. The framework will include a limited set of gestures that can be analyzed in a short time interval (essentially, two consecutive updates of the input sensor i.e., 60ms). These gestures can be combined into new ones. For example, we could combine the moving of both open hands to create a new gesture similar to *ExpandClose*. We will show how this can be achieved in section 4.3.

3 PetShop and ICO Models

PETSHOP is a tool developed at the ICS Lab of Toulouse for the specification, execution and verification of interactive critical systems. For many years, ICS has been developing dependable user interfaces with the help of formal methods. PETSHOP is actually used to create user interfaces for the next generation of Airbus aircrafts cockpits. PETSHOP uses the formal modeling language of Interactive Cooperative Objects (ICO) [13], which is based on high-level Petri nets [11] to describe the behavioral aspects of the interactive system to be designed.

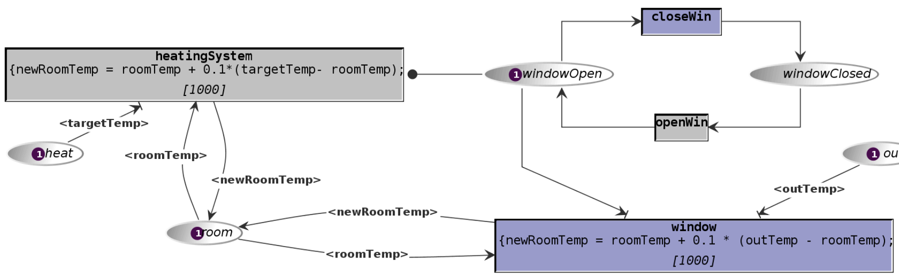


Fig. 2. ICO model of a room temperature control system

An example of an ICO model is shown in Fig. **2**. Data is represented by *tokens* that transit between *places* (represented by labeled ellipses) through *transitions* (labeled rectangles, connected to the places through incoming and outgoing arcs containing an arrow symbol at the end to specify their direction). A small numbered circle indicates the number of tokens in a nonempty place. Tokens can carry information in the form of Java objects with fields and operations. When a transition is fireable its color is changed to purple (it is gray otherwise). A condition can also be specified to act as a guard for a transition, and operations can be performed on the object stored in the tokens consumed by the transition. In addition to normal transitions, ICO models can contain *inhibitor arcs* (lines with a small circle at their endpoint, connecting the transition to a single place) that only allow the transition to fire if the place to which the arc is linked does not contain any token. *Test arcs* (represented by incoming arcs with a perpendicular small line just behind the arrow symbol) represent special transitions that do not consume the token used to trigger the transition.

Ordinary transitions consume tokens from incoming places and produce new tokens in outgoing places. These transitions are visualized as rectangular depressed buttons. *Synchronized transitions* are transitions that have to wait for an event in order to fire. These transitions are presented as rectangular pressed buttons. In Fig. **2** all transitions are synchronized: *openWin* and *closeWin* have to be triggered by the user, while *heatingSystem* and *window* are time-event transitions that are triggered every 1000ms. Fireable transitions are shown in purple.

Fig. **2** presents a small example of an ICO model representing a room temperature control system. It is composed of five places. A place labeled *heat* stores the target room temperature in a token *targetTemp* containing a Java object of type float. A place labeled *room* stores the actual room temperature in a token *roomTemp*. Place *out* stores the outside temperature in a token *outTemp*. A token is created in place *windowOpen* (resp., *windowClosed* if the user opens (resp., closes) the window. Four transitions connect these five places. User-triggered transitions *openWin* and *closeWin* cause the token referring to the state of the window to travel between *windowOpen* and *windowClosed* or conversely. Time-triggered transition *heatingSystem* (triggered every 1000ms) uses token *targetTemp* stored in place *heat* and consumes token *roomTemp* stored in place *room* to produce a new token *newRoomTemp* with the new temperature of the room, but only if the window is not open (modeled by an inhibitor arc to place *windowOpen*). Time-triggered transition *window* is triggered every 1000ms by a test arc from place *windowOpen* and another test arc from place *out*. If the window is opened, token *outTemp* is used and token *roomTemp* is consumed by the transition to produce a token *newRoomTemp* that replaces the original token *roomTemp*.

Initially, place *heat* contains one token *targetTemp*=25, place *room* contains one token *roomTemp*=18, place *out* contains one token *outTemp*=5 and place *windowOpen* contains one token with no associated value. When executing this model with PETSHOP, the room temperature changes every second, slowly decreases and converges to 5. If the user triggers the *closeWin* transition, the heating system turns on and the room starts to heat up, until it reaches the *targetTemp* temperature. The user can interact with the control system by opening or closing the window, and he can observe the temperature of the room evolve over time in the *room* place.

Although not used in the example, an important feature of ICO is that multiple models can communicate through events contained in a transition. These events can carry a Java object, providing a useful and intuitive way of transferring data from one ICO model to another. When a specific event is triggered in a transition, it is received by all ICO models connected to the sending model by an Observer design pattern.

4 Architecture of the Layered PetriNect Framework

We have chosen to develop our gestural interaction framework as a layered client-server architecture (see Fig. **3**). Input devices such as the Kinect implement a simple client connected to a Java server that is located, as an entry point, in the lowest part of the framework. The received information is then forwarded to the first layer of the framework. All framework layers are modeled as ICO models in PETSHOP. Raw data coming from the gestural input device through the Java server are sent to layer 1, which processes the information and sends it further to layer 2 and so on. Information that has been processed in layer 4 is sent outside the framework to the target application using the client-server architecture.

We will validate our framework through two case studies, using the Kinect as input device. The Kinect comes along with the NITE framework, providing a set of algorithms to perform real-time body tracking. These algorithms are used to retrieve the position of the hands and head of the user in 3D space, the origin of the coordinate system being located on the eye of the Kinect's camera. As the provided algorithms do not allow knowing whether hands are open or closed, we implemented this feature to widen the range of possible gestures to be recognized (see section 4.3 for the list of supported gestures).

On top of our framework, target applications can be developed using different output devices. To demonstrate the versatility of our approach, we have used a different output device for each case study. The first case study uses a Java Swing user interface, and the second uses C++ in combination with the Ogre3D graphical rendering engine.

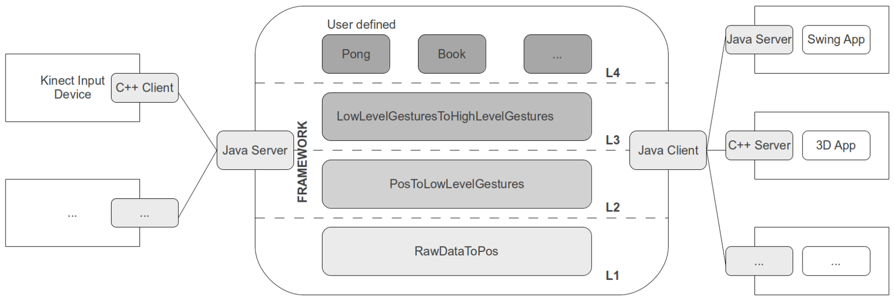


Fig. 3. Layered client-server architecture of the gestural interaction framework

4.1 Layer 1: Converting Raw Data into Positions

The goal of Layer 1 is to detect new users and propagate them through the rest of the framework, but also to receive raw information from the input device about the hands and head of the current users and combine them to calculate the hands' positions relative to the head. Every time such new information is available, an event is sent to Layer 1. Synchronized transitions *HeadEvent_* (the underscore at the end is a naming convention to indicate that this transition receives an event from outside the model) and *HandEvent_* are fireable, as they are only waiting for such an event to trigger, carrying the same name as the transition. The synchronized transition then generates a token, depending on which body part changed its position. The token generated by the transition contains positional information about the updated body part and the *id* of the user to whom the body part belongs.

Every 30ms, layer 1 receives and updates information about the users' positions. The ICO model of Fig. 4 contains two places for this purpose: *handsSet* stores one token per hand and per user, and *headsSet* stores a different token for the head of each user. When a new user id is encountered for the first time, transition *emptySetHands* is fired, and an event is sent to the next layer through transition *raise_NewUser*[3]. If

[3] We use the notation *raise* as a convention for transition that send an event to another model.

the user's id is already known, transition *CompareHeads* (resp. *CompareHands*) is fired. It replaces the old position of the head (resp. hand) with the newly received ones.

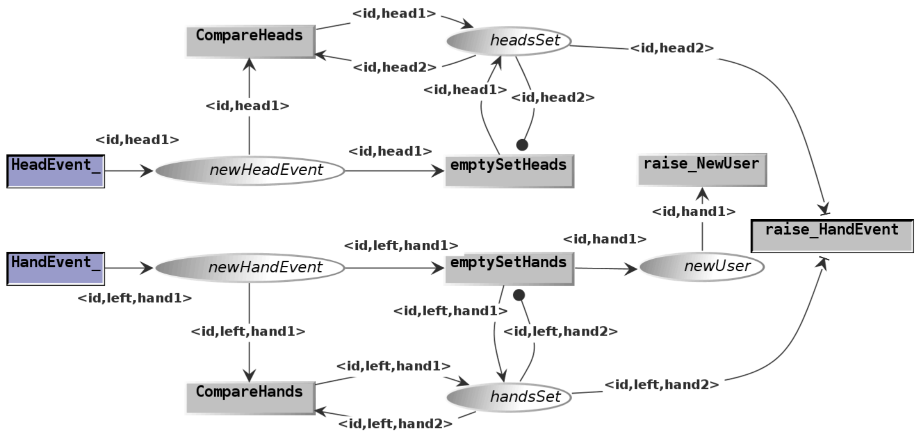


Fig. 4. Layer 1: ICO model converting raw input data into absolute positions.

This replacement is achieved using label matching (called *unification*) on arcs. For example, transition *CompareHeads* has two incoming arcs with labels <*id, head1*> and <*id,head2*>, respectively. In order to be fireable, this transition requires a token in place *newHeadEvent* with the same *id* as another token in place *headsSet*. As the second part of the label (*head1* and *head2*) does not match, they are considered to be different variables that can store different values. Unification is a very powerful feature of the ICO notation and is widely used throughout our layered framework. We will not further detail all unifications used; the attentive reader can observe their use by inspecting the depicted ICO models.

Periodically, one hand token and one head token sharing the same user id are consumed by the transition *raise_HandEvent*. This transition computes the position of the hand relative to the head using the code shown below. If the hand is located above the head then an *UP* label is associated to it. If the hand is located somewhere between the head and the hips then the *NORMAL* label is associated. Finally, if the hands are lower than the hips a *LOW* label is associated to the hand. This information can be useful depending on the type of designed interaction. For example one could consider that if the hands are not in *NORMAL* state, the gestures should not be interpreted.

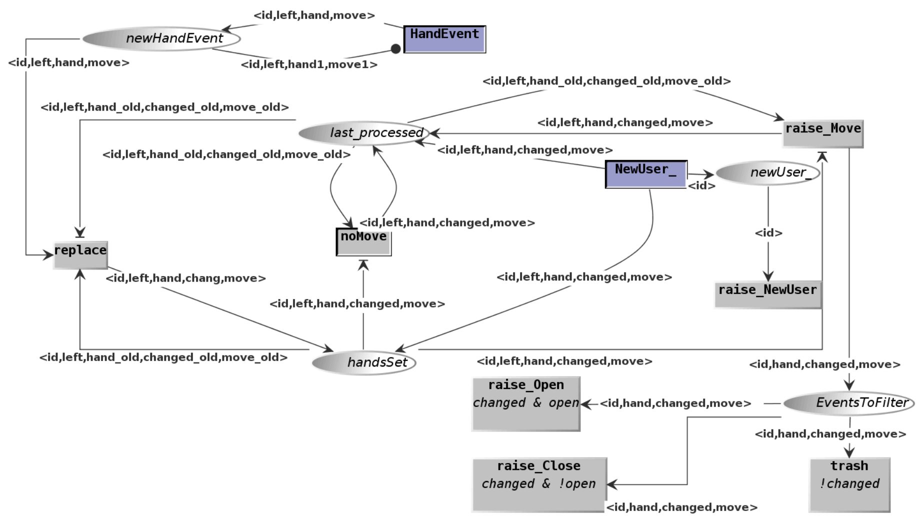
```
dist = head2.y - hand2.y;
if(dist<0) hand2.pos = bodyParts.HandEvent.Pos.UP;
else if(dist < 500) hand2.pos = bodyParts.HandEvent.Pos.NORMAL;
else hand2.pos = bodyParts.HandEvent.Pos.LOW;
trigger("HandEvent", new EventObject(hand1));
```

After associating this additional information to the hand, the transition raises an event *HandEvent* that is sent to Layer 2 (by means of the message *trigger* in the code snippet). This is done for all possible hand-head pairs belonging to each user.

Due to space considerations we will no longer show any code associated to transitions in the remainder of this paper. It should be noted that all models used in this paper are fully executable.

4.2 Layer 2: Transforming Positions to Low-level Gestures

Layer 2, shown in Fig. **5**, transforms absolute positions of the hands in space to relative movements between two consecutive updates. It is in this layer that state changes of the hands (opened or closed) are detected. When a new user is detected, transition *NewUser_* fires and creates a token in place *handsSet* while triggering a *NewUser* event.



<id,left,hand,move>
HandEvent
newHandEvent
<id,left,hand,move>
<id,left,hand1,move1>
<id,left,hand_old,changed_old,move_old>
<id,left,hand_old,changed_old,move_old>
<id,left,hand,changed,move>
raise_Move
last_processed
<id,left,hand_old,changed_old,move_old>
<id,left,hand,changed,move>
NewUser_
newUser_
<id>
<id,left,hand,changed,move>
noMove
<id>
replace
<id,left,hand,chang,move>
<id,left,hand,changed,move>
raise_NewUser
<id,left,hand,changed,move>
<id,left,hand_old,changed_old,move_old>
handsSet
<id,left,hand,changed,move>
<id,hand,changed,move>
raise_Open
changed & open
<id,hand,changed,move>
EventsToFilter
<id,hand,changed,move>
raise_Close
changed & !open
trash
!changed
<id,hand,changed,move>

Fig. 5. Layer 2: transforming the absolute positions into low-level gestures.

Events received from Layer 1 through synchronized transition *HandEvent_* generate tokens in place *newHandEvent*. Transition *replace* consumes the token containing the newly received position of the hand as well as the last matching processed token (in place *last_processed*), in order to compute the relative movement of the hand and its potential state change (open or closed). The unification is made on the *id* and *left* labels, to ensure that the two consecutive tokens must belong to the same user and refer to the same hand in order to make transition *replace* fireable.

Once the relative hand movement and state change is computed, they are stored as a token in place *handsSet*. Each time a new token arrives in this place, transition *raise_Move* can be fired to generate *Move* events. In addition, a token is created in place *EventsToFilter*. If the state of the hand has changed since the last processed frame, an *Open* or *Close* event can be triggered in places *raise_Open* or *raise_Close*, depending on the new state of the hand. If no change occurred in the hand's state, the token should be deleted through transition *trash*.

All 4 aforementioned generated events (*NewUser*, *Move*, *Open* and *Close*) are sent to Layer 3 together with information about the user id, the relative movement between two consecutive frames, and the updated left or right hand movement.

4.3 Layer 3: Combining Low-level into High-level Gestures

Layer 3 serves to combine the low-level gestures of Layer 2 (*Open*, *Close*, *Move*) into high-level ones that will be interpreted by virtual objects in Layer 4. Layer 3 is divided into 2 models (shown in **Fig. 6** and **7**) having 4 places in common (*leftClosed*, *rightClosed*, *twoOpened* and *twoClosed*). The first model serves to process the state of the users' hands. The second model combines low-level gestures into high-level ones by taking into account the state of the user's hands (e.g., both hands open). More models can be added to define other high-level gestures according to the users' needs.

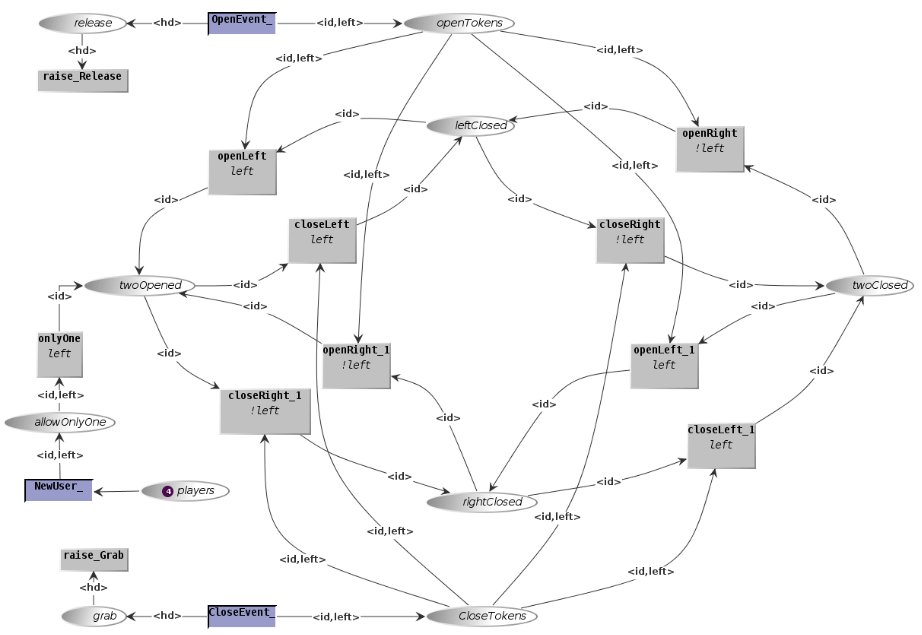


Fig. 6. Layer 3: Model processing the state of the users' hands.

Fig. 6 models the state of both hands of each user, that can either be open or closed. They are represented by 4 different places: *twoOpened*, *twoClosed*, *leftClosed*, *rightClosed*. (For the latter two states, we assume that the other hand is opened.)

Transition *NewUser_* processes the incoming event *NewUser* to consume a token from place *players* and associates an *id* to it when a new user joins the game. By default, the system initializes a new user with both hands open, and will produce a token in place *twoOpened*. When an *Open*$_2$ (resp. *Close*$_2$) event is received from Layer 2, transition *OpenEvent_* (resp. *CloseEvent_*) is triggered, which will produce a new token in place *openTokens* (resp. *closeTokens*). Note that we use subscript "$_2$" in the text to distinguish events coming from Layer 2 from events generated in Layer 3 itself.

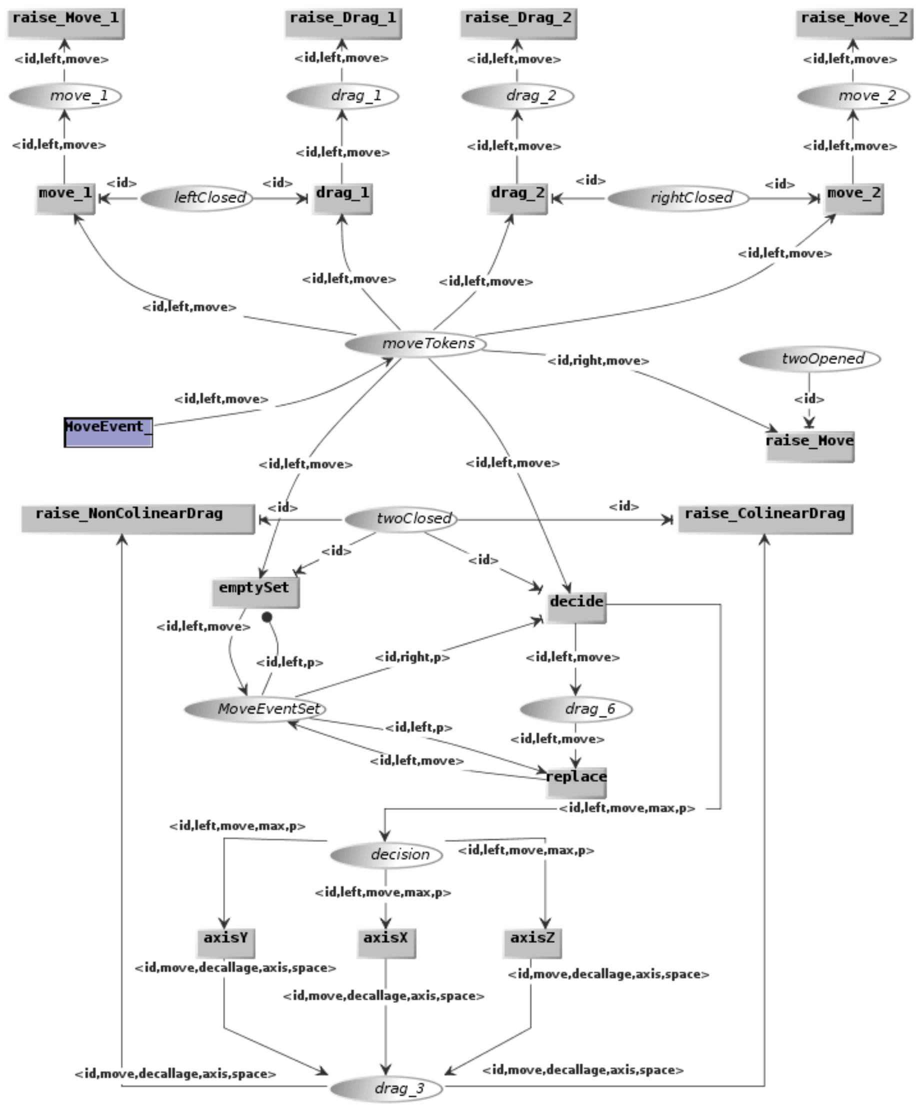


Fig. 7. Layer 3: Combining low-level gestures into high-level gestures.

Regardless of the user's state, *Close* and *Open* events are automatically generated by transitions *raise_Close* and *raise_Open* when a hand is closed or opened. Depending on the state of the user, one out of 8 possible transitions can be fired to change the user state. Imagine that a user enters the game with both hands opened. A token will be generated for his id in place *twoOpened*. If he closes the left hand, a *Close*$_2$ event is received and the transition *CloseEvent_* is fired, thus producing a token in *closeTokens* place. The only transition that can then be fired is *closeLeft*. This transition consumes both tokens and produces a new token for the user's id in place *leftClosed*.

The second part of Layer 3, depicted in Fig. **7**, combines low-level gestures into 4 high-level gestures: *Move*, *Drag*, *ColinearDrag* and *NonColinearDrag*. All of these rely on the incoming event *Move*$_2$, received from Layer 2, which triggers to *MoveEvent_* transition. This transition produces a token in place *moveTokens*. Depending on the user's state, different transitions can then be fired. If the user had both hands opened, transition *raise_Move* will be fired and generate a *Move* event. If the left (resp. right) hand was closed, then the resulting gesture will be either *Drag* or *Move* depending on which hand moved. If the closed hand moved, a *Drag* event will ultimately be generated. If the opened hand moved, a *Move* event will ultimately be generated.

If both hands were closed when a *Move*$_2$ event was received, then the gesture will be interpreted as a *ColinearDrag* or a *NonColinearDrag*, depending on a more complex calculation of the relative position of the moving hand w.r.t. the other hand. These calculations are done in the lower part of Fig. **7**, in the sub-model connected to place *twoClosed*. The idea is to use the *Move*$_2$ events and the positions of both hands to check wether the hands are colinear or not.

Another way to specify high-level gestures in Layer 3 is by combining a series of low-level and/or high-level gestures. For example, *GrabAndPull* would be a high-level gesture specified as a combination of a *Close* gesture and a consecutive series of *Drag* gestures along the z-axis. Such a high-level gesture could be used, for instance, to open a virtual door.

4.4 Layer 4: Manipulating Virtual Objects Through Events

Layer 4 differs from the previous layers in that it contains a different model for each type of object in the virtual scene the user needs to interact with. These objects can be controlled using the gestures provided by Layer 3. The same gesture can be interpreted differently by different objects, e.g., picking up an object lying on a desk or opening a door. By combining the provided gestures, one can create a wide range of behavioral models for an unbounded number of different types of objects. This versatility is illustrated through the case studies of Section 2.

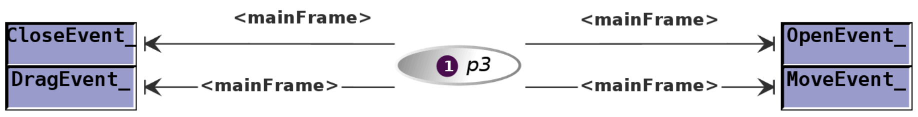


Fig. 8. Layer 4: Transferring gesture events to the virtual paddle object of the Pong game.

For the first case study (the Pong game), we have created an ICO model (shown in Fig. **8**) to represent the interaction with the Pong paddle. The virtual paddle object reacts to 4 types of gestures: *Close, Open, Drag* and *Move*. The model is very simple because the gestures do not need to be further processed. Their stored information can be directly used in the 3D application, such as the *id* of the player who is moving the hand (to move the right player's paddle) or the position of the hand itself. When *Move* is received, the paddle will move according to the gestures of the hand, by updating its position in the game. When *Drag* is received, the paddle still has to move, but can also respond to a *Close* event, followed by a *Open* event some time later. It is the responsibility of the game itself to decide how to interpret these events, e.g., checking if the ball is close enough to the paddle and moving the paddle along with the ball. We have chosen not to model the whole game with ICO models in this article, because our focus is on the interaction framework.

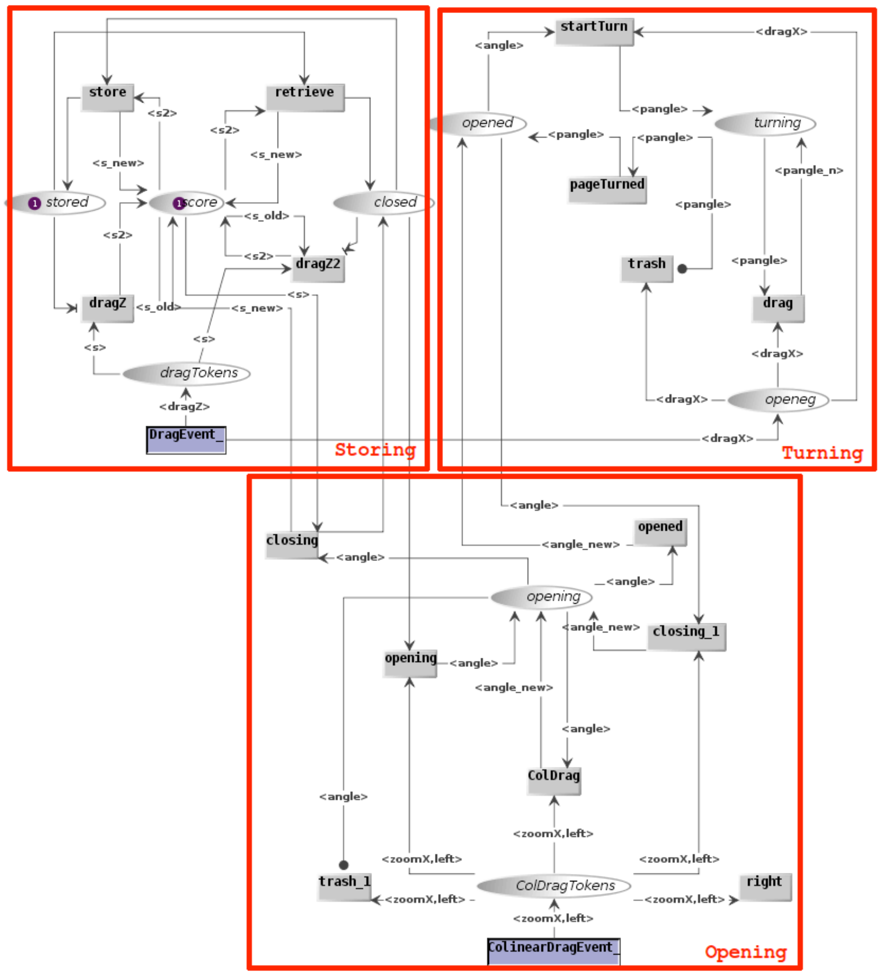


Fig 9. Layer 4: ICO model of the interaction behavior for a virtual book.

The gestural interaction of the second case study is shown in the ICO models of Fig. **9** and **10**. Fig. **9** represents the different book states together with the transitions between these states. For example, when a book is *stored* on the shelf, the user can only use *Drag* events to retrieve it. As we don't want to trigger the retrieval of the book at the first encountered drag, we designed a simple heuristic based on a *score* place. A token in place *score* is incremented by 1 each time a new drag on z-axis is performed and decremented by 3 when the drag goes in the opposite direction or if another gesture is encountered. When the score reaches a certain threshold, the *re-trieve* transition is fired and an event is sent to the rendering engine. This heuristic is used to avoid false positives. We want to trigger the retrieval of the book only after a certain amount of drags along the z-axis. To open the book and turn the pages, we use direct feedback. Each time a *ColinearDrag* (resp. *Drag*) event is received, the book (resp. page) flips a bit further, until a given threshold is reached and the book (resp. page) will be fully opened (resp. turned). Closing the book is modeled in a similar way.

Fig. **10** represents the interaction of the ICO model with the Ogre3D graphical rendering engine. All transitions from place *ogreClient* send messages to the rendering engine in order to give visual feedback to the user. All the transitions of this model can be found in Fig. **9**. These transitions are not fireable as there is not a token in each incoming place.

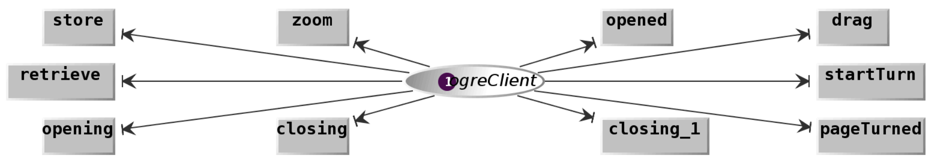
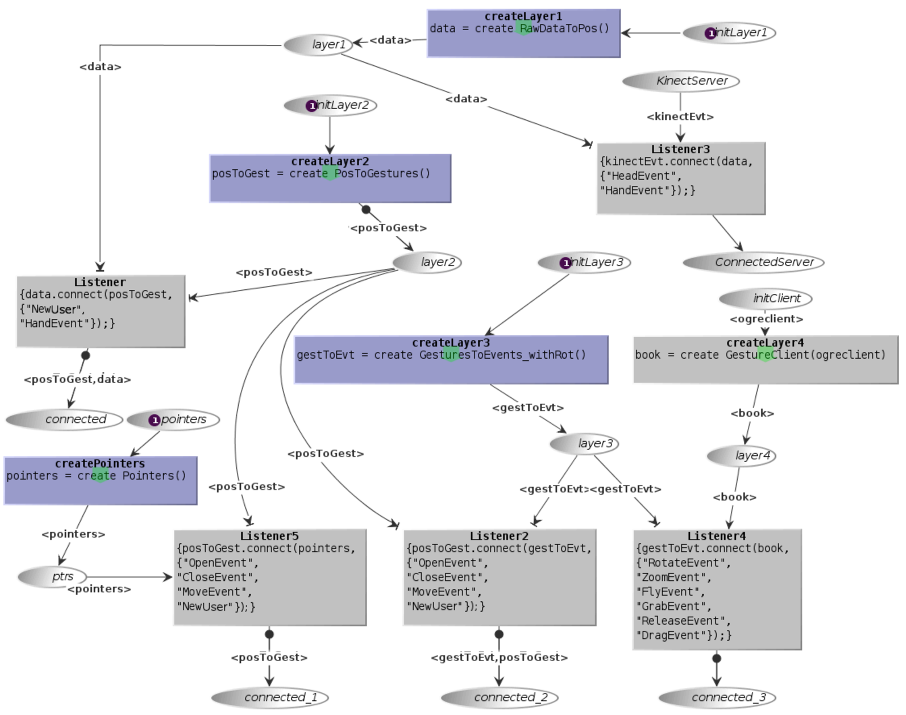


Fig. 10. Layer 4: Transitions of the book model connected to the Ogre3D Client

When multiple objects compose the scene, it is not always straightforward to know to which object the user's gestures should be applied. To solve this problem, we created an ICO model for a 3D pointer, presented as a simple hand that can be either opened or closed. This pointer is used to give visual feedback to the user. By drawing the position of the 3D pointer in the 3D rendering window, the user sees what virtual object is the closest to his hand so that when gestures are made, they will be applied to the closest virtual object. In our framework, the pointers model is connected to Layer 2, as it just needs the position, state and amount of movement of the hands.

4.5 Connecting the Layers

Communication between layers is explicitly modeled by use of an Observer design pattern [8]. This model, depicted in Fig. **11**, is executed only once at startup and is used to create the layers and establish the communication between them. It allows each higher layer of our architecture to receive and process events sent by the lower layers. Transitions containing the *create* primitives (depicted with a small green arrow inside the transition) consume a token to create a new model instance. For example, transition *createLayer1* produces a token containing a reference to the model of Layer 1 and puts it in place *layer1*. Similarly, transition *createLayer2* produces a token referring to the model of Layer 2 and puts it in place *layer2*. The models of both layers are connected by transition *Listener* specifying which events generated by the model of Layer 1 will be observed (i.e., received) by the model of Layer 2. Here, the events are *NewUser* and *HandEvent*. The same pattern is used to connect the other layers.


```
createLayer1
data = create RawDataToPos()

createLayer2
posToGest = create PosToGestures()

Listener3
{kinectEvt.connect(data,
{"HeadEvent",
"HandEvent"});}

Listener
{data.connect(posToGest,
{"NewUser",
"HandEvent"});}

createLayer3
gestToEvt = create GesturesToEvents_withRot()

createLayer4
book = create GestureClient(ogreclient)

createPointers
pointers = create Pointers()

Listener5
{posToGest.connect(pointers,
{"OpenEvent",
"CloseEvent",
"MoveEvent",
"NewUser"});}

Listener2
{posToGest.connect(gestToEvt,
{"OpenEvent",
"CloseEvent",
"MoveEvent",
"NewUser"});}

Listener4
{gestToEvt.connect(book,
{"RotateEvent",
"ZoomEvent",
"FlyEvent",
"GrabEvent",
"ReleaseEvent",
"DragEvent"});}
```

Fig. 11. Model connecting the layers of the framework using the Observer pattern.

5 Lessons Learned and Related Work

We can derive a number of lessons learned from using a visual formalism based on Petri nets for developing a gestural interaction framework and having applied it to two case studies. The visual formalism reduces the technical complexity of developing gestural interaction behavior by raising the level of abstraction. We have personally experienced that hardcoding such behavior using standard textual programming languages can be quite cumbersome. A visual formalism facilitates system understanding as its behavior is highly concurrent within models and as several models evolve in parallel. The dynamic simulation facilities allow finding and correcting bugs more easily and offer an elegant way of bridging the gap between modeling and execution.

When executing the models we can actually see the tokens moving from one place to another, and see their values changing as different transitions are fired. With PETSHOP, we can even change models during their execution, and study how this affects the running process. The fact that ICO is formally defined and that it is grounded in Petri nets theory makes it possible to exploit formal verification techniques to assess and prove properties over a set of cooperating models. We have not presented these capabilities due to space constraints but more details can be found in [13].

One very useful additional feature was the ability of models to communicate by means of events, using synchronized transitions in the listening models to trigger transitions only when an event is received. This enabled the creation of a layered architecture in which behavior can be added or modified easily. One way to do so is to modify each layer individually. Another possibility is to add new layers on top of the existing ones to extend the set of recognized gestures, by combining the existing set of gestures. The label unification capability of ICO models allowed us to implement multi-user functionality in our framework in a very straightforward way, without the need to change the structure of our models. Each user is represented by a different token, and the token *id* is matched when processing the transitions to ensure that the information of the correct user is being considered (for example in case of coordinated hand and head movement). Thanks to the ability to store data in tokens, we were able to process hand state and position in a very straightforward way. This data could be used for calculations in the transitions thus resulting in an increased expressiveness. The notion of test arcs allowed us to create store places that always contain the latest update about the hands position/state. The tokens could be used without being consumed, which was very useful since we wanted to keep these tokens until the next update (to compute relative movement for instance). Inhibitor arcs were also used frequently to deal with special cases such as the very first token regarding a hand (to initialize the store places).

The work presented in this article has convinced us that executable visual modeling is the way to go for developing interactive applications. Not only is it adapted for this kind of systems, but also the learning time is relatively low. Without any prior knowledge of Petri nets or ICO models, it took the first author about 4 weeks to learn the formalism and the PETSHOP tool, and to design a first running version of our framework. After that, it turned out to be very easy to add more functionality (such as pointers and new gestures) to our models. Excluded from the aforementioned learning time is the time needed to understand how to use and interface with the gestural input and graphical output devices, as they fall outside the scope of the framework.

A relevant question is whether ICO is the most appropriate notation for specifying executable gestural interaction models. Other visual notations could be used instead. For example, [7] used *statecharts* to manipulate virtual objects displayed on screen using hand movements. It is difficult to compare the effectiveness of that approach with the current one, since the statecharts were hardcoded in Java using the SwingStates API [1]. Although it would be possible to achieve the same result using executable visual statechart models [11], the ability to add or remove new users or virtual objects dynamically would be more difficult to achieve.

Another alternative for executable specification of gestural interaction is the use of *heterogeneous modeling*. [6] explored this alternative by developing a layered client-server framework using the ModHel'X environment [10]. The different layers were specified using different visual formalism such as *Synchronous Data Flow models* [9] and *Timed Finite State Machines (TFSM).* The challenge of heterogeneous modeling lies in the need for semantic adaptation between the different layers using different formalisms, as well as the need to master multiple formalisms.

A limitation of our approach is the potential difficulty for domain experts and designers of interactive applications to master the notation of ICO models. It is important to note, however, that the low-level models are device-dependent and only have to be built once. Designers then only need to adapt the models to implement the desired behavior for the target interaction technique. Beyond that, we believe that the full expressive power of ICO is not always needed and can be abstracted away by using a domain-specific modeling language. At the risk of a certain loss of expressiveness, this would allow designers to compose gestures and process them to create new behaviors for virtual objects more easily, without requiring detailed knowledge of the underlying visual formalism.

Another current limitation is the performance issues encountered during our case studies, because PetShop interprets and simulates ICO models at runtime. To be able to apply it in real commercial applications, faster execution is necessary. This can be achieved by compiling the models directly into executable code, but it will go at the expense of no longer being able to dynamically visualize and modify the models during their execution, which is very useful for development and debugging purposes.

In [14], ICO models were proposed to model multimodal interactions in virtual reality applications. As a proof-of-concept, a virtual chess game was developed that could be manipulated by a single user using a data glove on one hand. The design of this application was quite different from ours. It did not focus on reusable models, did not include the state of the user or the notion of virtual objects, and contained a very limited set of gestures using only one hand.

Many other tools and flavors of high-level Petri nets exist. *Flownets* are an alternative that has been used for modeling virtual environments [16]. Based on the authors' observation that virtual environments are made up of a complex combination of discrete and continuous processes, the particularity of flownets relies in the combination of discrete and continuous behavior to specify the interaction with virtual environments. Resorting to continuous models is a possible solution, but it may not always be the most appropriate, especially in presence of sensors that provide discrete events at a variable frame rate, depending on the performance of the computer used.

According to [17], one of the limitations of many virtual environment toolkits is the predefined and limited small amount of interaction means they provide, which are intended to be used regardless of context. To extend the flexibility of such toolkits, developers must be provided with the possibility to design, test and verify new interaction techniques. The authors presented *Marigold*, a toolset supporting such a development process. This toolset allows visual specification of the interaction techniques (using flownets) but unlike our approach, it is not dynamic. C code is generated from the models, making it impossible to modify the models at runtime or to see the models running . Moreover, the entire specification resides in a single monolithic model, unlike our layered architecture that is more modular and easier to adapt.

In the context of modeling for interaction, [5] proposed Interface Object Graphs, a method based on statecharts dedicated to the specification and design of new interaction objects or widgets. In contrast to our approach, the user was not explicitly modeled. In addition, the interaction was object-centric, making it difficult to specify multimodal interaction with multiple users.

6 Conclusions and Future Work

We presented PETRINECT, a model-based approach for developing interactive applications allowing the execution of visual models and relieving the developer from writing complex and statically compiled code. PETRINECT is a generic, layered and modular framework to specify and execute gestural interaction based on the visual formalism of high-level Petri nets. Our case studies illustrated how to use PETRINECT to allow a user to interact with multiple virtual objects displayed on screen. The client-server architecture of our framework allows developers of interactive systems to easily integrate gestural interaction in their future entertainment projects. Other usage scenarios of our framework are conceivable, such as remotely controlling devices (e.g., domotics, multimedia).

Our case studies have shown the feasibility of visually specifying executable models for real-life applications. These models are based on discrete events provided by the gestural input devices. High-level Petri nets, incarnated as ICO models in PETSHOP, proved to be particularly suited for expressing such models. One of the reasons was their ability to concurrently execute complex behavior involving a dynamically changing number of actors (e.g., players, virtual objects) and requiring a huge amount of user interaction. Using high-level Petri nets allowed us to cope with the high complexity by separating the behavior of gestural interactions in separate communicating layers. In addition, data manipulation and data flow were facilitated by the ability to encapsulate data in tokens.

Future work will be carried in the directions mentioned in section 5. We plan to validate the framework further using other input devices such as the Wiimote, and we plan to define a domain-specific language on top of ICO as a mean to provide the scaffolding required for wider use by designers and developers.

Acknowledgments. We thank Alexander Serebrenik, Gilles Geeraerts and Frédéric Boulanger for giving us feedback on a draft version of this article. The first author is financed by a FRIA scholarship. This research has been partially supported by research project AUWB-08/12-UMH-3 “Model-Driven Software Evolution”, an Action de Recherche Concertée financed by the Ministère de la Communauté française – Direction générale de l’Enseignement non obligatoire et de la Recherche scientifique, Belgium.

References

1. Appert, C. and Beaudouin-Lafon, M.: SwingStates: Adding state machines to Java and the Swing toolkit. Software Practice and Experience, 38(11):1149–1182 (2008)
2. Bailly, G., Walter, R., Müller, J., Ning, T. and Lecolinet, E.: Comparing free hand menu techniques for distant displays using linear, marking and finger-count menus. In Proc. INTERACT, pp. 248–262 (2011)
3. Bolt, R. A.: "Put-that-there": Voice and gesture at the graphics interface. In Proceedings SIGGRAPH '80, pp. 262-270. ACM, New York (1980)
4. Boulos, M.K.N., Blanchard, B., Walker, J.M.C. and Tripathy R.G.-O.A.: Web GIS in practice X: a Microsoft Kinect natural user interface for Google earth navigation. International Journal of Health Geographics, 10:14 (2011)
5. Carr, D.A.: Specification of interface interaction objects. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 372–378, ACM (1994)
6. Deshayes, R., Jacquet, C., Hardebolle, C., Boulanger, F. and Mens, T.: Heterogeneous modeling of gesture-based 3D applications. In MoDELS Workshop on Multi-Paradigm Modeling (2012)
7. Deshayes, R. and Mens, T.: Statechart modelling of interactive gesture-based applications. In First International Workshop on Combining Design and Engineering of Interactive Systems through Models and Tools (ComDeisMoto), satellite event of INTERACT (2011)
8. Gamma, E., Helm, R., Johnson, R.E., and Vlissides, J.M.: Design patterns: Elements of reusable object-oriented software. Addison-Wiley (1994)
9. Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D.: The synchronous dataflow programming language Lustre. Proceedings of the IEEE, 79(9):1305–1320. IEEE (1991)
10. Hardebolle, C. and Boulanger, F.. ModHel'X: A component-oriented approach to multi-formalism modeling. In MoDELS Workshops, LNCS 5002, pp. 247–258. Springer (2008)
11. Harel, D. and Gery, E.: Executable object modeling with statecharts. Computer, 30(7):31–42. IEEE (1997)
12. Jensen, K. and Rozenberg, G., editors. High-level Petri nets: theory and application. Springer-Verlag, London, UK (1991)
13. Navarre, D., Palanque, P., Ladry, J.-F. and Barboni, E.: ICOs: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. ACM Trans. Comput.-Hum. Interact., 16(4):18:1–18:56. ACM (2009)
14. Navarre, D., Palanque, P.A., Bastide, R., Schyn, A., Winckler, M., Nedel, L.P. and Freitas, C.M.D.S.: A formal description of multimodal interaction techniques for immersive virtual reality applications. In INTERACT, LNCS 3585, pp. 170–183. Springer (2005)
15. Ren Z., Meng J., Yuan J., and Zhang Z.: Robust hand gesture recognition with Kinect sensor. In ACM Multimedia, pp. 759–760 (2011)
16. Smith S. and Duke D.: Virtual environments as hybrid systems. In Proceedings 17th Annual Conference Eurographics UK, United Kingdom (1999)
17. Sukaviriya, P. N., Kovacevic, S., Foley, J. D., Myers, B. A., Olsen Jr., D. R. and Schneider-Hufschmidt, M.: Model-Based User Interfaces: What Are They and Why should we casre? In Proceedings UIST'94, November 1994, pp133-135; ACM DL
18. Willans, J.S. and Harrison M.D.: A toolset supported approach for designing and testing virtual environment interaction techniques. Int. J. Hum.-Comput. Stud., 55(2):145–165 (2001)