Heterogeneous Modeling of Gesture-Based 3D Applications

Romuald Deshayes[*] Christophe Jacquet[†] Cécile Hardebolle[†]
Frédéric Boulanger[†] Tom Mens[*]

[*] University of Mons
Mons, Belgium
firstname.name@umons.ac.be

[†] Supelec E3S – Computer Science Dept.
Gif-sur-Yvette, France
firstname.name@supelec.fr

ABSTRACT

Model-driven software engineering (MDE) is a well-known approach for developing software. It reduces complexity, facilitates maintenance and allows for the simulation, verification, validation and execution of software models. In this article, we show how MDE and model execution can be leveraged in the context of human-computer interaction (HCI). We claim that in this application domain it is beneficial to use heterogeneous models, combining different models of computation for different components of the system. We report on a case study that we have carried out to develop an executable model of a gesture-based application for manipulating 3D objects, using the Kinect sensor as input device, and the OGRE graphical engine as output device for real-time rendering. The interaction part of this application is fully specified as an executable heterogeneous model with the ModHel'X modeling environment. We exploit the semantic adaptation between different models of computation to implement a layered application using the most appropriate models of computation for each layer.

Categories and Subject Descriptors

H.5.2 [**Information Systems**]: Information Interfaces and Presentation—*User Interfaces*; I.4.9 [**Computing Methodologies**]: Image processing and computer vision—*Applications*; I.6 [**Computing Methodologies**]: Simulation and Modeling

General Terms

Design, Languages, Experimentation, Human Factors

Keywords

model-driven software engineering, heterogeneous modeling, human-computer interaction, gestural interfaces, ModHel'X

1. INTRODUCTION

Developing human-computer interaction (HCI) applications and 3D applications can be a challenging task for programmers for various reasons. It requires a lot of domain-specific knowledge, and it involves a high level of technical complexity, introduced by the many technical and hardware-related programming issues. This accidental complexity [5] can be reduced by resorting to off-the-shelf frameworks and libraries for natural interaction (e.g., OpenNI[1] and NITE[2]), computer vision (e.g., OpenCV[3]) and real-time rendering (e.g., OpenGL[4], OGRE[5] and JMonkey Engine[6]), and by wrapping them through simple-to-use APIs so that the HCI application developer is insulated from the technical details.

The accidental complexity can be reduced even further by resorting to *executable* models that specify the behavior of interactive 3D applications. The use of models promises several benefits: the task of the developer will be facilitated by raising the level of abstraction; and the models can be simulated and verified in order to detect conceptual problems in the application earlier in the life-cycle.

We advocate the use of *heterogeneous* models to allow the HCI developer to use the most appropriate models at the most appropriate level of abstraction, and to seamlessly integrate these different models into an executable whole. By heterogeneous modeling we mean that different models of computation are used for the different components that make up the application. For example, one component could be modeled using discrete events, another one using synchronous data flow, yet another one using timed finite state machines, and all of these components need to interact and communicate together in a seamless way. The ModHel'X modeling environment permits this, by using hierarchical heterogeneity and explicit semantic adaptation between heterogenous parts of a model [3].

As a case study to illustrate how modeling provides an added value in the context of HCI application development, we report on the development of an executable model of a gesture-based 3D application using the ModHel'X environment for heterogeneous modeling [8]. The heterogeneous model of our HCI application offers several benefits to the application developer: (i) the application-specific behavior can be specified visually and changed easily at a very high level of abstraction, without the need for writing program code; (ii) the gestural input received from the user's body parts can be interpreted and manipulated independent of, and without being aware of, the technical specificities of the input format; (iii) the virtual 3D objects that reside in

[1] Open Natural Interaction – `openni.org`
[2] `primesense.com/nite`
[3] Open Source Computer Vision – `code.opencv.org`
[4] Open Graphics Library – `opengl.org`
[5] Open source Graphics Rendering Engine – `ogre3d.org`
[6] `jmonkeyengine.com`

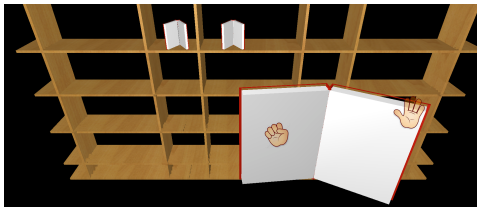

Figure 1: 3D book manipulation.

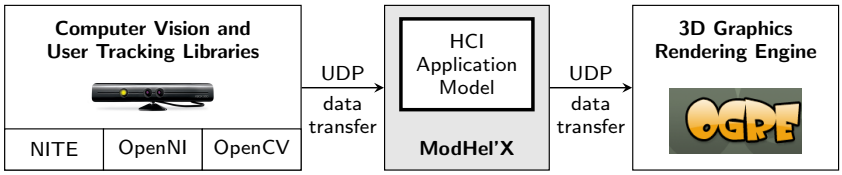

Figure 2: Architecture of the approach.

the graphical library can be integrated and reused easily in different applications, since the behavior of these objects is specified or customized by the application developer using visual models.

The remainder of this article is structured as follows. In Section 2 we introduce our HCI case study, relying on the Kinect sensor and the OGRE graphical rendering engine. Section 3 then presents ModHel'X, the environment we selected for heterogenous modeling, and the adaptations that were required to use it for our case study. Section 4 explains how the case study has been modeled in ModHel'X, and motivates our design choices. Section 5 presents related work. In Section 6 we discuss our proof-of-concept and present avenues of future work, before concluding in Section 7.

2. CASE STUDY: BOOK MANIPULATION

As a case study, we consider a simple example of a gestural HCI application to interact with 3D graphical objects on the screen. The goal is to allow a user standing in front of the screen to interact with virtual books by using his hands. We envision the following application scenario. The user can open a virtual 3D book displayed on the screen by swiping his left hand from right to left. Conversely, by swiping his left hand from left to right, he will close the book. *Swiping* a hand is achieved by first closing the hand and then moving it in the given direction. The user can also move the book around in 3 dimensions while his right hand remains closed. Figure 1 shows a screenshot of the application implementing this scenario.

Depending on the scenario of use, the developer can easily customize the behavior of the HCI application and the behavior of the graphical objects that are used through the use of visual modeling:

- He can attach a specific interpretation to the hand gestures. In the presented scenario, the left and right hand act differently: the book can only be opened or closed while the left hand is closed; the book can only be moved while the right hand is closed.
- He can attach a different behavior to the same graphical object (in our case: the book). In the presented scenario, moving, opening and closing the book is allowed. Other actions, like rotation, are disallowed.

Figure 2 depicts the global architecture of the HCI application. Only the functional logic of the application (middle part of the figure) is expressed as a heterogenous model in ModHel'X. It sends data over UDP to a graphical engine (right part, hand-coded in C++) to render 3D objects on the screen, and it receives data over UDP from an input device (left part, hand-coded in C++) to detect hand gestures. For the former, we used the open source OGRE rendering engine to specify the graphics of the application. For the latter, we use Microsoft's *Kinect* controller[7]. It detects the shape and 3D position of a person in front of the screen through the use of an infrared projector and an infrared camera. We used *NITE*, a C++ development framework for the *Kinect* to detect persons in front of the sensor and to track the 3D position and movement of their skeleton in real time. For our HCI application, we only use the position of each hand (in 3 dimensions) as well as its status (open or closed). Since the information of whether a hand is opened or closed was not provided by the NITE framework, we hand-coded this functionality ourselves in C++.

We now present the ModHel'X framework, before detailing the heterogeneous model used in our HCI application.

3. HETEROGENEOUS MODELING WITH MODHEL'X

ModHel'X [8, 3] is an experimental framework developed at Supélec in order to test new ideas about the executable semantics of heterogeneous models. There are two main tasks to achieve in order to obtain a meaningful heterogeneous model using model composition: (1) the precise definition of the semantics of each modeling language; (2) the precise definition of the semantic adaptation between parts of a model that use different modeling languages. One method for defining the semantics of different modeling languages is to use a common meta-model to describe the structure of models, and to attach semantics to this structure using so-called *models of computation (MoC)*.

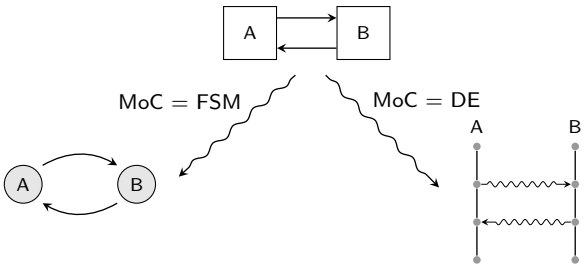

Figure 3: Models of computation.

A MoC is a set of rules that define the nature of the components of a model and how their behaviors are combined to produce the behavior of the model. For instance, Figure 3 shows that two models can share the same structure (two components A and B linked by two arrows) with different semantics, i.e., different MoC: a finite state machine (FSM) or two processes communicating through discrete events (DE).

[7] `www.xbox.com/kinect`

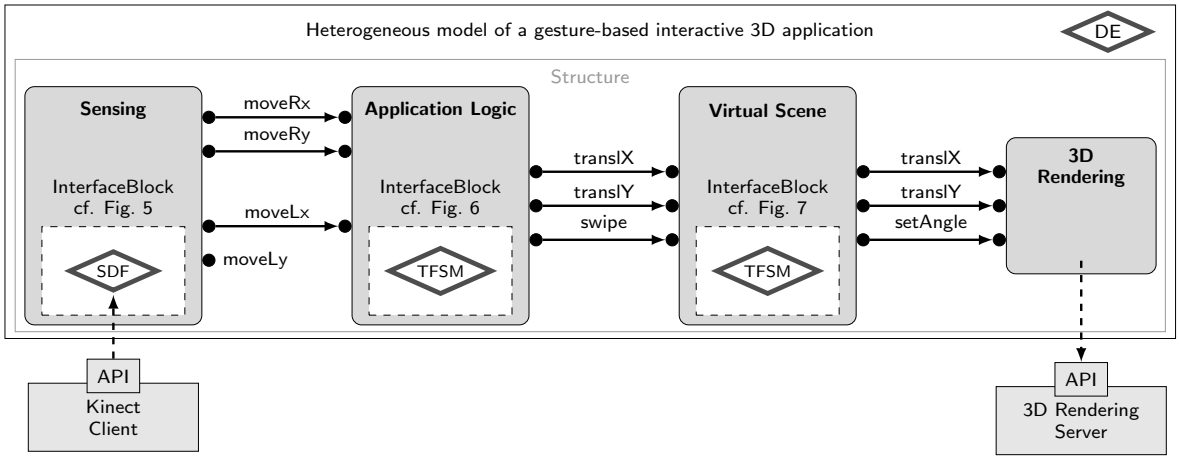

Figure 4: Structure of the heterogenous model for the "Book manipulation" case study.

When interpreted by the FSM MoC, the model represents an FSM with two states. When interpreted by the DE MoC, it represents two processes that communicate through events.

ModHel'X allows one to describe the structure of heterogeneous models, to define MoCs for interpreting such structures, and to define the semantic adaptation between heterogeneous parts of a model. For this, ModHel'X relies on a meta-model that defines the common abstract syntax for all models, and on a generic execution engine that defines an abstract semantics that is refined by MoCs into a concrete semantics.

Figure 4, which shows the structure of the ModHel'X model used for our case study, illustrates the different elements of the abstract syntax of ModHel'X. A model is composed of a structure (surrounded by a gray outline), which contains *blocks* (the gray rectangles with rounded corners), and which is interpreted according to a MoC (shown in a diamond-shaped label). In this example, the model is interpreted according to the DE MoC, and contains four blocks: Sensing, Application Logic, Virtual Scene, and 3D Rendering. Blocks are considered as black boxes that communicate only through *pins* (the black circles). The structure of a model is defined by setting *relations* (the arrows) between pins.

In ModHel'X, interpreting a model means executing the behavior described by that model according to the semantics of the MoC. An execution is a series of observations of the model, each observation being computed through the sequential observation of the blocks of the model using a fixed-point algorithm. The observation of one block is called an *update*. Each MoC dictates the rules for scheduling the update of the blocks of a model, for propagating values between blocks, and for determining when the computation of the observation of the model is complete.

Examples of MoCs currently available in ModHel'X include Timed Finite State Machines (TFSM), Petri nets, Discrete Events (DE) and Synchronous Data Flow (SDF). TFSM and Petri Nets are well-known MoCs. DE and SDF work like their implementations in Ptolemy II [4, 7]: in DE, blocks are processes that exchange timestamped events that can contain data; in SDF, blocks are data-flow operators that consume and produce a fixed number of data samples on their pins each time they are activated.

In ModHel'X, heterogeneity is handled through hierarchy. A ModHel'X model can contain *InterfaceBlocks*, whose internal behavior is described by a ModHel'X model. The MoC used by the inner model of an interface block can differ from the MoC of the outer model to which the interface block belongs. The *InterfaceBlock* acts as an *adapter* between the two MoCs. For instance, in the detailed view of the Application Logic block shown in Figure 6, the MoC of the outer model is DE, and the MoC of the inner model is TFSM. The dashed arrows between the pins of the interface block and the pins of the inner model represent the *semantic adaptation* between the two MoCs, which is realized by the interface block. As shown in [3], three aspects can be considered in this adaptation: *data* (which may not have the same form in the inner and outer models), *time* (the notion of time and the time scales may differ in the inner and outer models) and *control* (the instants at which it is possible or necessary to communicate with a block through its interface).

4. CASE STUDY REVISITED

This section details how we used ModHel'X to specify an executable model of the case study that was introduced in Section 2. A webpage presenting this model and videos of its execution by interacting with a user through the Kinect is available at `wwwdi.supelec.fr/software/ModHelX/Kilix`.

Figure 4 shows the overall structure of our gesture-based interactive 3D application: the Sensing block receives data from the Kinect and converts it into hand gesture events; the Application Logic block interprets these hand gestures and converts them into actions that are meaningful in the context of an interactive 3D application; and the Virtual Scene block represents graphical 3D objects (e.g., a book) that interpret these actions, convert them into object-specific behavior (such as opening or closing the book), and send instructions to the 3D Rendering block that communicates with the graphical rendering engine.

We have chosen the discrete events MoC (DE) for the communication between the top-level blocks. Submodels may use other MoCs. The Sensing model uses synchronous data flow (SDF), the Application Logic and Virtual Scene models use timed finite state machines (TFSM).

The internal model of the Sensing block is shown in Figure 5. It relies on data packets received from the Kinect device at a fixed rate. It is a signal processing chain acting upon

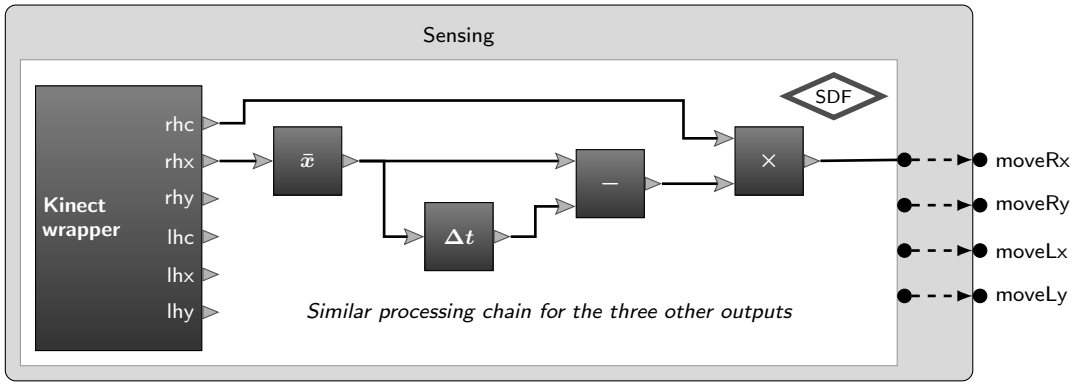

Figure 5: Interface block and SDF model of the Sensing block to detect hand gestures.

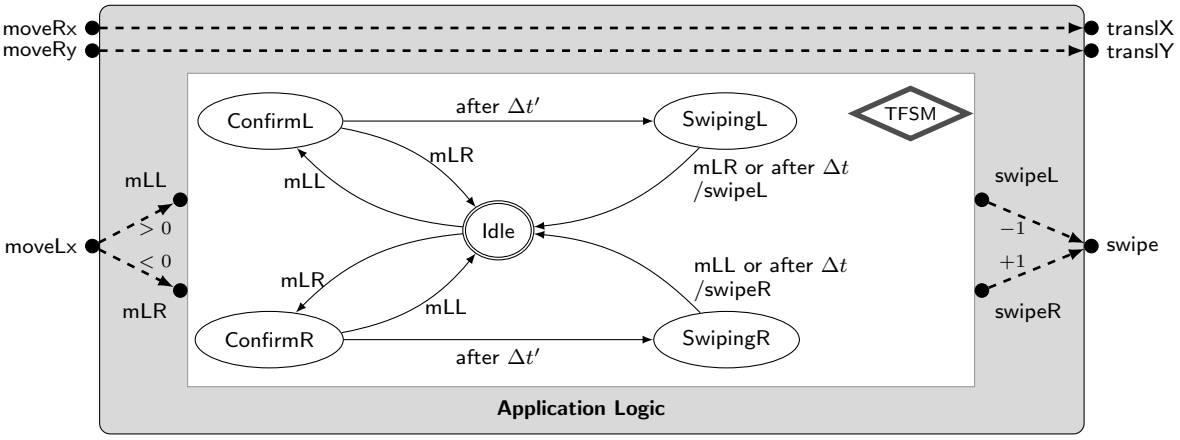

Figure 6: Interface block and TFSM model for the Application Logic block.

sampled signals, therefore SDF is the most appropriate MoC to use. A wrapper around the Kinect interface provides the position (x, y coordinates)[8] of each hand (left and right) and its status (open or closed). This SDF model transforms the absolute positions into relative movements such as moving the left hand (resp. right hand) horizontally or vertically. For each x and y component of each hand, it first averages the coordinate to eliminate high-frequency fluctuations. Then it calculates the relative moves by subtracting a delayed coordinate from the current one. Finally, only movements performed while the hand is closed are taken into account, so we multiply the amplitude of the movements by the hand status (0 stands for "open"; 1 stands for "closed"). On output, we generate DE events only for the movements that have a non-null amplitude. For this, we reuse the pre-existing SDF/DE adapter, customized in order to generate DE events only when non-null SDF tokens are produced. In the DE model, these events are then passed to the Application Logic block for further processing.

Figure 6 shows the Application Logic block containing the main functional logic of the interactive application. Its inner model uses the timed finite state machines (TFSM) MoC. As we have shown in previous work [6], the use of state machines provides a natural way to represent and implement the application logic of interactive gesture-based applications. The purpose of the application logic is to recognize some application-specific gestures such as swiping to the left and to the right, or moving graphical objects around. First, the discrete events received from the Sensing block are converted by the adapter into state machine events. For instance, when the left hand moves along the x coordinate (incoming `moveLx` event) with $\Delta x < 0$ then an event called `mLL` (*move left hand to the left*) is sent to the state machine. An `mLR` event is generated likewise. The state machine detects these elementary movements and aggregates them into swiping movements after they have been performed for a while. On output, the state machine produces `swipeL` and `swipeR` messages that are converted back into DE events by the interface block.

To achieve this, we had to extend the original DE/TFSM adapter in two ways. First, we needed to be able to generate FSM events based on *conditions* such as $\Delta x < 0$, whereas previously we could only discriminate between discrete values. This improvement paves the way for the support of an *expression language* in ModHel'X. Second, some events (e.g., `moveRx` and `moveRy`) just needed to be passed on by the Application Logic. Instead of adding transitions to the state machine, which would make it unnecessarily difficult to read and maintain, we added the ability to connect output pins directly to input pins in the interface block (the two dashed arrows at the top of Fig. 6).

The state machine of Fig. 6 works as follows. When nothing happens, it stays in the initial `Idle` state. As soon as an `mLL` event is received (representing the movement of the

[8]For brevity we did not process z coordinates.

left hand to the left), a transition to the `ConfirmL` state is fired. If during a short time interval (represented by $\Delta t'$) no movement in the opposite direction is detected, the left hand movement is confirmed and a transition goes to the `SwipingL` state. After a fixed time delay Δt, or if the hand moves to the right, the `swipeL` message is generated and the state machine returns to the `Idle` state. The behavior for the right hand is symmetrical (with a `ConfirmR` state and a `SwipingR` state, an `mLR` event that triggers the transition and a `swipeR` message that is generated).

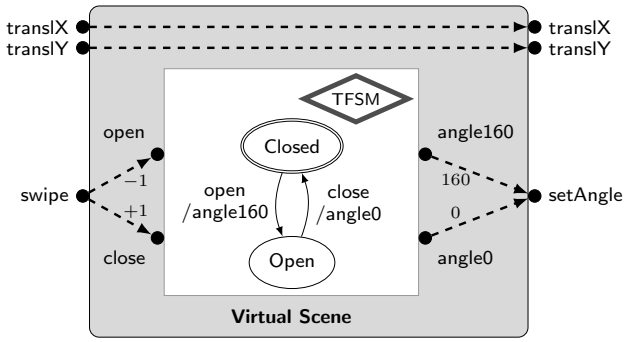

Figure 7: Interface block and TFSM model for the Virtual Scene block.

The Virtual Scene block of Fig. 7 represents the virtual 3D object (the book) the user will interact with. This block specifies the behavioral model of the interactions with the virtual object, using the TFSM MoC. When the book is in the `Open` state, and the `close` event is received (corresponding to a positive `swipe` event received from the outer block), a transition to the `Closed` state is triggered, and an `angle0` event is generated to tell the graphical rendering engine to set the angle between the two covers of the book to 0 degrees. The rendering engine will respond to this by an animation that closes the book. If the book is in the `Closed` state, we have the opposite behavior. When an `open` event is received (corresponding to a negative `swipe`), a transition to the `Open` state is triggered, and an `angle160` event is created to tell the graphical rendering engine to open the book by setting the cover to cover angle to 160 degrees. The `translX` and `translY` events are not interpreted by the TFSM, but are simply forwarded by the DE/TFSM adapter in a similar way as for the Application Logic block.

Having presented this new approach of the modeling of HCI applications, we will compare it in the next section to other approaches before discussing its limitations.

5. RELATED WORK

In the field of modeling for HCI, our previous work [6] used only Statecharts to model the gestural interaction with a 3D user interface using the Kinect. We developed a simple framework to represent and execute various user interactions such as pointing, clicking or drag and drop, and also translating or rotating graphical 3D objects. In the same field, Navarre et. al. [9] used refined Petri nets to model the interaction with a virtual chess game enhanced with a data glove, thus allowing direct manipulation of chess pieces. They defined a formalism called Interactive Cooperative Objects (ICO) [10] based on Petri nets to specify, model and execute interactive systems. The main difference between our current work and the aforementioned references resides in the fact that we now use heterogeneous modeling whereas a homogenous formalism was used previously. Heterogeneous modeling allows us to use different modeling languages for different parts of the application. For instance, we feel that processing raw 3D coordinates from the Kinect is better described using data flow operators, and state machines are a better fit for describing the behavior of the application.

Regarding the choice of ModHel'X as the heterogeneous modeling environment, we could also have used Ptolemy II [7], Metropolis[1] or the MATLAB/Simulink toolchain by The MathWorks [9]. However, MATLAB supports only a restricted set of modeling languages, while Ptolemy and ModHel'X provide support for the creation of new modeling languages. This is an advantage because a modeling environment for HCI applications should support various domain specific modeling languages depending on the application domain and on the properties that should be checked on the application. The advantage of ModHel'X over Ptolemy II is that the semantic adaptation between heterogenous parts of a model is explicitly defined. In this case study, we saw that semantic adaptation, which was initially thought of as a mechanism for adapting between different modeling languages, can also be used to adapt between different uses of a component in various applications. For instance, the interface block used to embed a book in the virtual scene not only adapts the semantics of DE to the semantics of TFSM, it also maps actions (`swipe`, `translate`) to reactions of the book (`open`, `close`, `move`). This mechanism eases the reuse of components by decoupling the description of their behaviors from the events that trigger these behaviors.

6. DISCUSSION AND FUTURE WORK

Currently, the application handles only one book and one user. The book is modeled as a state machine in a ModHel'X block. A real-scale application would require to deal with a whole bookshelf and multiple simultaneous users, so it would become tedious to add many identical ModHel'X state machines and blocks to the overall model. To ease the construction of such a model we would need some "syntactic sugar" to instantiate an array of several nearly-identical blocks. In this regard ModHel'X currently has another limitation: models are static, i.e., their structure cannot change at runtime; only their execution context evolves over time. Handling a variable number of books and users would require the ability to change the model at runtime. A possibility would be to add support for ModHel'X blocks that behave as classes instead of objects, and to model instances of those classes as data tokens. This way, a TFSM model would represent the behavior of all books, and this behavior would be applied to a particular book when a data token representing this book would be put on an input pin of the block. On output, the setAngle pin, for example, would hold a data token identifying both the 3D object to update and the value of the angle between its covers.

We are currently carrying out an extended case study to compare the homogenous Petri net approach proposed by ICO [9, 10] with the heterogenous approach presented here. This will allow us to explore the added value of using concurrency and dynamic instantiation for HCI applications.

In the present proof-of-concept we used dedicated models

[9] `www.mathworks.com/products/simulink/`

of computation (MoCs) for each part: for instance a MoC for gesture recognition, a MoC for specifying library management tasks, a MoC for interfacing with the 3D rendering, etc. In ModHel'X, a MoC and its associated library of blocks corresponds to a domain-specific language. The use of very specialized DSLs for specific tasks is an advantage for the designer of an application, but it poses the problem of defining and tooling these DSLs. Creating a MoC and a library of blocks in ModHel'X requires some non trivial work. Therefore, we are currently working on an approach for leveraging the ModHel'X notion of MoC and meta-modeling techniques to ease the definition of new DSLs by reusing models of computation and libraries of domain-specific actions. By allowing the designer to work with domain-specific concepts and by reducing the semantic gap between the application domain and the modeling tools, this approach will be beneficial to the modeling of gesture-based 3D applications.

Finally, when developing interactive applications, an architectural and design model has to be chosen. In our case, we have used a derivative version of the Arch architecture [2]. Thanks to this model, we are able to distinguish and represent each architectural component of our case study, such as the step that involves the transformation of low-level events to high-level actions, and the input-device independent step which models the interaction with graphical 3D objects and interprets the high-level actions.

7. CONCLUSIONS

In this paper we have successfully used the ModHel'X modeling environment for developing a simple HCI application for gesture-based user interaction. The proof-of-concept application we developed served several purposes: (i) to illustrate the feasibility of developing HCI applications in a model-driven way, limiting the amount of code to be developed by the HCI programmer to the bare minimum; (ii) to assess the usefulness of heterogeneous modeling for this purpose, using different model components with different models of computation; (iii) to identify the use of semantic adaptation as a mechanism for reusing objects in different application contexts; (iv) to overcome some of the limitations of ModHel'X and suggest future improvements.

Because we could directly simulate our case study in ModHel'X, we were able to develop our HCI application using rapid prototyping. It only took us 3 full days to realize the application. In addition, our modular architecture made it significantly easier to modify the application along two dimensions: (i) to change the gestural input device (Kinect in our case) or the graphical output rendering engine (OGRE in our case), only one single component needs to be replaced; (ii) to change the application-specific behavior or the object-specific (books in our case) behavior, again only one single component needs to be replaced. Ultimately, it should be possible to come up with a reusable library of graphical object models, each having their own specific behavior that can be customized by the user when integrating them into the HCI application.

8. ACKNOWLEDGMENTS

We thank Hans Vangheluwe and Pieter Mosterman for bringing us together, through the organization of the CAMPaM 2012 workshop on Multi-Paradigm Modeling.

This research has been partially supported by (i) the F.R.S.–FNRS through FRFC project 2.4515.09 "Research Center on Software Adaptability"; (ii) research project AUWB-08/12-UMH "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. The first author is financed by a FRIA scholarship.

9. REFERENCES

[1] F. Balarin, L. Lavagno, C. Passerone, A. L. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. Modeling and designing heterogeneous systems. In J. Cortadella, A. Yakovlev, and G. Rozenberg, editors, *Concurrency and Hardware Design*, volume 2549 of *Lecture Notes in Computer Science*, pages 228–273. Springer, 2002.

[2] L. Bass, R. Pellegrino, S. Reed, S. Sheppard, and M. Szczur. The arch model : Seeheim revisited. In *CHI 91 User Interface Developpers Workshop*, 1991.

[3] F. Boulanger, C. Hardebolle, C. Jacquet, and D. Marcadet. Semantic adaptation for models of computation. In *Proc. Int'l Conf. Application of Concurrency to System Design (ACSD)*, pages 153–162. IEEE, 2011.

[4] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java (volume 3: Ptolemy II domains). Technical Report UCB/EECS-2008-30, University of California, Berkeley, 2008.

[5] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 20th anniversary edition, 1995.

[6] R. Deshayes and T. Mens. Statechart modelling of interactive gesture-based applications. In *Int'l Workshop on Combining Design and Engineering of Interactive Systems through Models and Tools (ComDeisMoto), satellite event of INTERACT 2011*, 2011.

[7] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, and Y. Xiong. Taming heterogeneity - the Ptolemy approach. *Proc. IEEE*, 91(1):127–144, 2003.

[8] C. Hardebolle and F. Boulanger. Exploring multi-paradigm modeling techniques. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 85(11/12):688–708, November/December 2009.

[9] D. Navarre, P. A. Palanque, R. Bastide, A. Schyn, M. Winckler, L. P. Nedel, and C. M. D. S. Freitas. A formal description of multimodal interaction techniques for immersive virtual reality applications. In M. F. Costabile and F. Paternò, editors, *INTERACT*, volume 3585 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2005.

[10] D. Navarre, P. A. Palanque, J.-F. Ladry, and E. Barboni. Icos: A model-based user interface description technique dedicated to interactive systems addressing usability, reliability and scalability. *ACM Trans. Comput.-Hum. Interact.*, 16(4), 2009.