Software Support for Coherent Prototyping of 3D Gesture Interactions

Dominik Rupprecht(✉), Rainer Blum, and Birgit Bomsdorf

Fulda University of Applied Sciences, Leipziger Straße 123, 36037 Fulda, Germany
{dominik.rupprecht,rainer.blum, birgit.bomsdorf}@cs.hs-fulda.de

Abstract. When prototyping applications that include touchless 3D gesture interaction three design matters must be taken into consideration: the gestures the user must execute, the visual representation and the dialog flow. Ideally, these aspects should be considered in parallel, to achieve a coherent design process, and avoid ineffective extra effort stemming from coordination between them. A flexible changeover of perspectives among the separate matters is needed. This paper proposes a software environment that enables the desired coherent rapid prototyping of applications with 3D gesture interactions. Its core consists of two types of mapping and a so-called co-simulation functionality. The environment facilitates combining existing software tools from industry and literature to cover the three design matters, i.e. specification and simulation of UI prototypes, gestures, and dialog models. It assists developers at design time in the specification of gestures and in binding them to UI prototypes as well as to statecharts used for defining dialog models. Relevant coherency information is used to offer the option to evaluate gestures at runtime in the context of UI prototype and dialog model. The co-simulator manages the synchronized simulation of all relevant artefacts once a gesture event occurs. Therefore, it enables quickly building prototypes that go beyond the capabilities of the individual tools. This paper describes the usage of the proposed environment in form of a case study with several software tools, each covering one of the three design aspects. It also shows its general applicability, meaning that it can be used with other tools too.

Keywords: Gesture development · Interaction design · User interface design Model-based development · Development support

1 Introduction

When developing applications with touchless 3D gestures, three design matters and their mutual dependencies are important [2]:

- The gestures: The movements and/or static poses of the human body or parts of it required to execute an intended interaction with a system must be defined.
- The presentation: The user interface must be designed and developed that informs the user about the required gestures and presents the system feedback.

- The dialog: The dialog flow specifies the interactions and their sequences, e.g. the conditions and point in time a specific gesture invokes a system functionality.

Bomsdorf and Blum [2] state that the design of gestures, presentation and dialog span a coherent design space, with different involved stakeholders (different project participants like user interaction designers, end users or software developers). Each of them needs specific tool support for one up to all the design matters. Gestures appropriate to interact with the system must be identified and specified in detail as part of this process. They can be bound to the presentation (e.g. starting with simple UI prototypes) and to dialog models. After that connection, the prototype of an application can be evaluated iteratively. The three design aspects must be kept synchronized. Furthermore, an easy switching between the different axes of the design space is needed. For instance, if a designer alters gestures, this could also imply modifications of the presentation, to present suitable affordances to the user.

Software support of this coherent, three axes design space may be pursued by two kinds of approaches for enabling an easy switching between the axes of the design space. A single integrated tool can be a reasonable option if it combines functionalities to create UI prototypes and dialog models and bind gestures to them [2]. It has the advantage that users benefit from a consistent terminology and user interface typically present in a single software tool. However, the downside is that prospective users first need to familiarize with such a new design tool and put their usual tools aside. This may impede the acceptance of the new tool. Also, due to the fact, that gesture sensors are not yet usable via plug and play, such single integrated tools must be altered for every new gesture sensor to be added. In addition, as industrial practice and the literature show, state of the art tools for these different design spaces exist and are widespread and established. Therefore, this work proposes a solution that integrates these existing software tools with the aim to support a coherent rapid prototyping of 3D gesture interaction and a flexible change of perspective between the three axes of the design space.

The next section presents related work focusing on the dialog axis and its combination with 3D gestures, as well as on the presentation axis and its combination with 3D gestures. This is followed by related work concerning the concept of co-simulation. Section 3 gives a broad overview of the software environment proposed in this paper and demonstrates the integration of the above-mentioned tools in form of a case study. Section 4 shows how the proposed environment is applicable to other tools or other interaction techniques. The paper concludes in Sect. 5 with a summary and discussion of possible future developments.

2 Related Work

2.1 Model Based Approaches for Dialog Modeling

In model-based approaches the focus is on the systematic development of an interactive systems. According to Meixner et al. [7] the "interface model" consists of all used models, divided into task, dialog, and presentation, while the dialog constitutes the central point. The latter describes the interaction of a user with the user interface and the resulting invocation of system functionality - largely independent of a specific technical implementation and look and feel. The concrete presentation is developed later. Often, Harel statecharts are used to specify dialog models (e.g. Feuerstack et al. [4]). Their central concepts are dialog states, transitions between these states, triggers set off by user actions that cause these transitions, and conditions to control them. Different modeling tools that implement this kind of statecharts are the Scade Suite[1], the Nutaq Model-Based Design Kit[2], or Yakindu[3], which is utilized in the work described in this paper. These modeling tools help designers to produce valid diagrams via automatic checks and auto-complete functions (cf. Kistner and Nuernberger [5]).

To prove the quality of the workflow of an application, user inputs can be simulated by triggering them in the UI of the modeling tool, while the reactions of the system are visualized with animated diagrams (cf. Pintér et al. [8]).

A connection of gestures with dialog models (working in the dialog-gesture design space) can be found in the work of Feuerstack et al. [4] and Bomsdorf et al. [3]. Both allow executable models to be controlled by gestures. But, dialog model and gestures are specified separately and involve the extra effort of an explicit modeling step that links the gestures to the executable dialog model, after the gestures have been implemented for a specific gesture recognizer.

With GestIT Spano et al. [9] show and categorize the difficulties to a model-based design of applications with gesture interaction. They propose a declarative and compositional framework for multiplatform gesture definitions as a step towards a new model to solve the single-event granularity problem and providing a separation of concerns. But their approach targets at later stages in the development process where applications are already created by programmers. Our goal is to enable prototyping of interactive gesture applications in earlier design phases.

2.2 Rapid Prototyping of Gesture Interaction

Rapid prototypes are an effective method if design ideas shall be implemented and evaluated with stakeholders like intended users or customers already in the design phase. These are visual approaches focusing amongst other things on the look and feel (structure and behavior) of the UI, in contrast to the above-mentioned presentation models, which are abstract UI specifications. In early stages authors like van Buskirk and Moroney [10] recommend working with paper prototypes or other low fidelity prototypes to test different ideas fast with little effort. As stated by Van den Berg et al. [11], once interactive behavior is needed, more sophisticated prototypes (e.g. implemented with HTML) are typically realized with prototyping tools like Balsamiq[4], Axure[5], or Pidoco[6]. These tools allow the creation of click-through wireframes to

[1] http://www.esterel-technologies.com/products/scade-suite/.

[2] http://www.nutaq.com/software/model-based-design-kit/.

[3] https://www.itemis.com/en/yakindu/statechart-tools/.

[4] https://balsamiq.com/products/mockups/.

[5] https://www.axure.com/.

[6] https://pidoco.com/en.

simulate an application. Balsamiq and Pidoco support touch gestures to be used in prototypes, but not touchless 3D gestures yet. The latter is one central topic of this work, and we incorporated Pidoco as example for our approach. Rapid prototypes focus on the perceptible presentation, while the dialog is typically given solely implicit in the quite simple form of the implemented sequence of user interface changes that can be caused by user actions.

2.3 Co-simulation

In an iterative, systematic and user-centric development the different design perspectives should be supported to an equal extent by a software tool or an environment. The interim results of each axis of the design space must be checked and their consequences understood in the other axes of the design space, due to their mutual concurrence representing all the same application (cf. Barboni et al. [1]).

An early approach to that challenge forms NVIDIAs UI Composer Studio[7]. This embedded systems tool, targeted at the automotive industry, allows designers to specify dialogs for digital dashboards and to bind them to the structure of graphical user interfaces. It relies on its own XML schema to bind the transitions of the dialog model to the expected structural ("page") changes in the presentation of the prototypes. According to Kistner and Nuernberger [5] NVIDIA UI Composer Studio eases both, the prototyping and the dialog modeling for complex applications, providing a coherent development while relying on a single tool.

The approach of Martinie et al. [6] faces a similar problem of enabling a coherent design and co-simulation of different separate aspects of an application. It binds an already existing application to a task model which is based on the already implemented workflow of the application. This is achieved through java annotations inserted within the program code of the application (changing the application program code itself). These annotations make it possible to extract the interactive widgets of the application automatically. After defining a systematic correspondence (a mapping) between these widgets and the task model, both, the application and the task model, can be co-simulated, i.e. they are executed synchronously. However, this approach is intended for later stages of the design process compared to our approach and it does not focus on gestures explicitly. In addition, the code of an application to be bound to a task model must be altered as described. But, that work is another example that shows the complexity of a bidirectional combination of models and presentation.

3 Software Environment

This section introduces our approach to support the required coherent design. The proposed solution utilizes existing, established tools for dialog modeling, rapid prototyping and gesture recognition and connects them with the help of a semantic-driven binding. Its purpose is to overcome the described shortcomings of the side-by-side use

[7] http://uicomposer.nvidia.com/.

of such tools, to take the mutual dependencies of the three design matters gesture, presentation and dialog into account and in sum to enable a coherent development of 3D gesture interactions.

To make the different tools work together closely, a mapping is needed. Mapping means, that semantically corresponded aspects of each tool are put into relation to each other, like the execution of a specific gesture and the associated reaction in a prototype. This forms the central prerequisite for the desired co-simulation. At design time a common set of triggers (called *sync triggers* further in this paper) is shared between the involved tools to realize the mapping and co-simulation.

3.1 Internal or External Mapping

Two types of mappings are implemented between the existing design tools to cater for the different cases that exist in practice: *Internal* and *external mapping*. If programmatic access to the internals of a program code is available, that can be used to import sync triggers and bind them to the internal control flow of the software, the mapping is implemented natively in the source code of the tool. We call this option *internal mapping*. Consequently, the actual communication between the tools is made possible via an application programming interface (API), that provides access to the required, mapping functionality. All tools must import a simple text file containing a list of *sync triggers* (trigger set) and map them to the desired internal functionality. If an event occurs in one tool that is mapped to a *sync trigger*, this trigger is subsequently called in the other tool via the involved trigger APIs.

For software tools with no or too limited access to the program code of the tool itself a concept called *external mapping* is used. This approach keeps the necessary changes on the software code as minimal as possible if at all. However, in this case the respective tool must feature an API that permits access to some kind of internal event system. That event system is bound to external events that are handled in other components of our environment using a separate *mapping editor*. The mapping between two tools is arranged with this editor during the design phase. As a prerequisite, the tool in question must provide a functionality to export information regarding its relevant internal events into a text file (a list of event names). These events are mapped to the *sync triggers* inside the *mapping editor*. In addition, for the intended co-execution of all involved tools a *co-simulator* is needed. During run time it processes the defined mappings by translating events outcoming from a tool into *sync triggers* and vice versa.

3.2 Environment Overview

In this section we detail our approach by means of a case study with the following software products as exemplary tools for the different axes of the design space:

- Presentation: Pidoco (see Footnote 6) for rapid prototyping of graphical UI.
  Pidoco is a client/server software tool with web front-end using up to date web technologies. It allows quick creation of click-through wireframes and fully interactive prototypes to simulate and test the look and feel of an application concept.

- Dialog: Yakindu (see Footnote 3) for building statecharts.
  Yakindu is a set of plugins written in Java within the Eclipse RCP and IDE environment[8] consisting of four main parts: statechart editing, simulation, validation and code generation.
- Gestures: Microsoft Kinect and a gesture simulator.
  The Kinect (Version 2)[9] is a combination of soft- and hardware for building and recognizing full body gestures. Instead, or in parallel, a gesture simulator can be used, which can trigger gesture events (as if done by the Kinect).

Internal and external mappings had to be realized. The Pidoco company allowed the native extension of their software tool to support *internal mapping* between *sync triggers* and the actions of the UI prototypes built with the Pidoco tool. The *external mapping* approach was used for the dialog modeling inside Yakindu since this tool features an extension API. A direct *internal mapping* of gestures was not desired by Yakindu. Therefore, this case study featured a combination of *internal and external mapping*. Figure 1 gives an overview of the case-specific implementation of our environment.

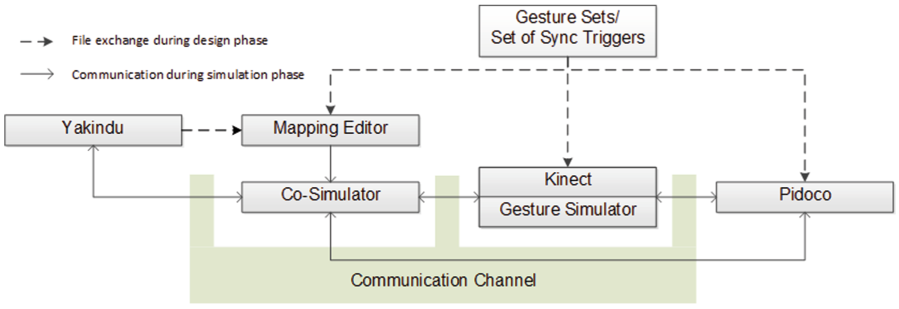

File exchange during design phase
Communication during simulation phase
Gesture Sets/ Set of Sync Triggers
Yakindu
Mapping Editor
Co-Simulator
Kinect
Gesture Simulator
Pidoco
Communication Channel

Fig. 1. Overview of implemented environment

In general, in the design phase a set of *sync triggers* must be compiled. Our *sync triggers* consist of a set of gestures, which are expected to occur, and they serve as a key data structure since all involved software tools are bound to it. For this implementation, this set of *sync triggers* (simple JSON file) was imported into the Kinect recognizer and in parallel into a *gesture simulator* software. This enabled the two tools to trigger gesture recognition events within the environment (resulting in the invocation of corresponding *sync triggers*).

Pidoco's native extension can import a *sync trigger* set (same JSON file), providing functionality to bind single *sync triggers* to reactions inside a UI prototype and, thus, to react to received triggers. Yakindu was extended with an export functionality for statechart events and variables via the extension of its existing API. When working on a

[8] https://www.eclipse.org/downloads/packages/eclipse-rcp-and-rap-developers/heliosr.

[9] https://developer.microsoft.com/en-us/windows/kinect.

concrete prototype, the user must import this statechart information (also a JSON file) into the *mapping editor*, where it must be bound to the *sync trigger* set by the user.

These mappings saved either inside a tool (*internal mapping*; in this case with the Pidoco tool) or externally in the mapping editor (*external mapping*; Yakindu, Kinect, gesture simulator), are subsequently used in the simulation stage for the synchronized execution of the tools. In our case study implementation, if a gesture is recognized by the Kinect gesture recognizer or is triggered manually within the *gesture simulator*, it is sent over through the shared communication channel. The Pidoco software tool listens for these events and then checks, if a prototype transition is mapped to an incoming *sync trigger*. It then reacts accordingly by carrying out the determined reaction. Each change during runtime within the UI prototype simulation caused by a user (e.g. a button click) is in turn propagated over the shared communication channel to all applications.

Based on the defined mapping between gestures and *sync triggers*, both structures are translated into one another by our *co-simulator*. A gesture, whether triggered by the Kinect recognizer (or, alternatively, the *gesture simulator*) is translated into statechart events and then sent to Yakindu. Each event occurring inside the statechart tool is sent to the network respectively. It is then translated back into gestures, respectively *sync triggers* by the *co-simulator* for the other tools to react.

In our case study the possibility to test and evaluate the interplay of prototype artefacts from two or three axes of the design space in parallel is demonstrated, using either real gestures or simulated ones. The next sections provide more details about the purpose of all parts of the environment including the implementation of required additional software components.

3.3 Gesture Processing

Gesture recognition was accomplished with a tool chain using Microsoft's Kinect (version 2). Typically, gestures are first recorded with the Kinect Studio[10] and then processed with the Visual Gesture Builder[11] resulting in a set of gestures. For our case study, these gesture sets were then extended with the required meta data including name, description of involved body parts and additional input values for the gesture recognizer. This editing task was implemented with the Kinect SDK in form of a separate component called *gesture editor,* where the *sync trigger* (JSON) file is created.

With our case study implementation, it is also possible to create gesture sets inside this *gesture editor* without recording actual gestures. These "virtual" gestures are composed of the mentioned meta data and a descriptive video demonstration. A *gesture simulator* can use both the "virtual" gesture sets, and the sets recorded with the Kinect to trigger *sync triggers*.

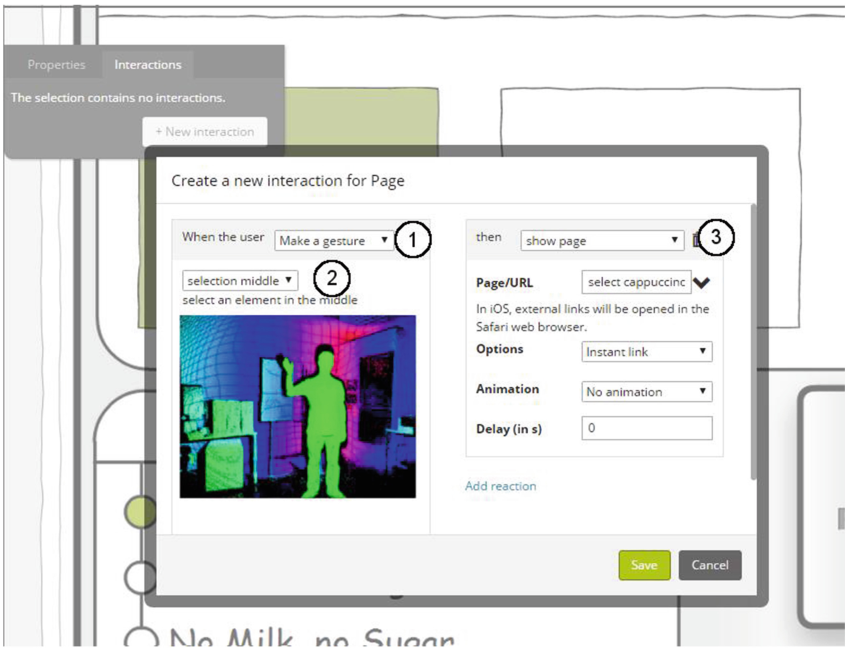
[10] https://msdn.microsoft.com/de-de/library/dn785306.aspx.

[11] https://msdn.microsoft.com/de-de/library/dn785304.aspx.

3.4 Mapping

Pidoco. As mentioned above, for the design of the presentation, i.e. a UI prototype, the mapping between gestures respectively *sync triggers* and actions within prototypes was implemented inside Pidoco (*internal mapping*). Actions like mouse clicks, touch gestures or motion of a mobile device can be configured as triggers inside Pidoco for user interface reactions without requiring any programming. These reactions range from UI page changes to playing sounds or highlighting areas. Each interaction consists of a user action and a system reaction. These interactions can be added via a context menu (see Fig. 2) to every interactive element on the screen. To support the *internal mapping* approach the Pidoco company extended this context menu, so that the array of assignable actions includes gestures as additional actions (see Fig. 2, ①).

To specify gestures as actions within a prototype a gesture set (a set of sync triggers) must be loaded via file import into Pidoco first (another functionality that was also added to the Pidoco software). The gestures are then available as actions in the aforementioned context menu. After selecting an action (see Fig. 2, ②) it can be bound to the desired reaction (see Fig. 2, ③).


Properties Interactions

The selection contains no interactions.

+ New interaction

Create a new interaction for Page

When the user: Make a gesture

selection middle

select an element in the middle

then: show page

Page/URL: select cappuccino

In iOS, external links will be opened in the Safari web browser.

Options: Instant link

Animation: No animation

Delay (in s): 0

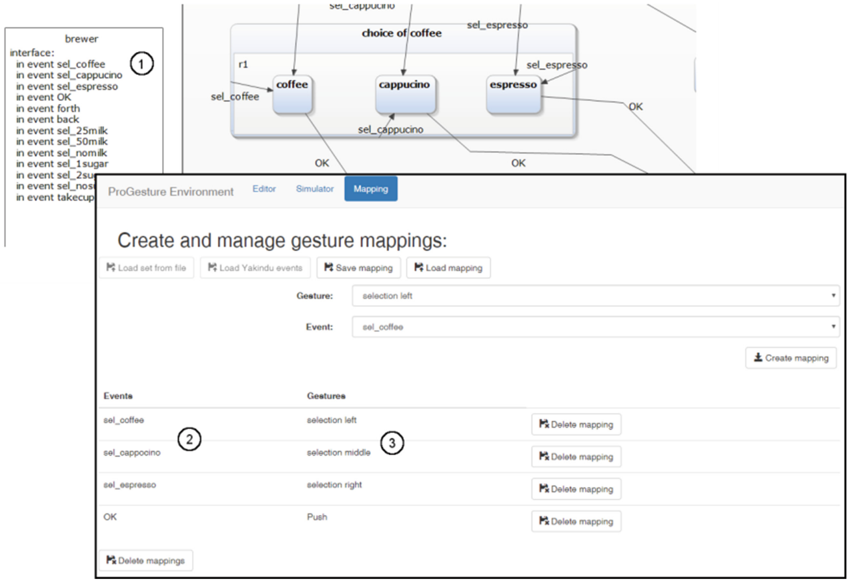
Add reaction

Save Cancel

Fig. 2. Mapping in Pidoco (extract)

Yakindu. For the design of the dialog, mappings must be created in the external *mapping editor* (*external mapping*). It maps gestures onto events within model state-charts. First, events are to be assigned to state transitions inside Yakindu. They trigger the transitions, representing dialog changes, if all defined conditions are met. The basic version of the Yakindu software is open source and provides a flexible extension interface, enabling the extraction and processing of design and runtime information.

To create the *external mapping*, information about the internal events and variables of a specific statechart must be exported. This export functionality was developed as a so-called decoupled code extension using the built-in features of the Yakindu API. The extension builds a JSON file automatically. All the events of a Yakindu statechart (see Fig. 3, ①) are represented in this JSON file. To establish the binding the user must import the file into the *mapping editor* inside our environment. The *mapping editor* then extracts the trigger information. Afterwards the Yakindu events (see Fig. 3, ②) must be mapped to the *sync triggers* (see Fig. 3, ③) by the user.



```
brewer
interface:
  in event sel_coffee
  in event sel_cappucino
  in event sel_espresso
  in event OK
  in event forth
  in event back
  in event sel_25milk
  in event sel_50milk
  in event sel_nomilk
  in event sel_1sugar
  in event sel_2su
  in event sel_nos
  in event takecup
```

Create and manage gesture mappings:

Gesture: selection left

Event: sel_coffee

Events	Gestures
sel_coffee	selection left
sel_cappocino	selection middle
sel_espresso	selection right
OK	Push

Fig. 3. External Mapping in Yakindu ① and the mapping table in the mapping editor (② and ③) (extracts)

3.5 Co-simulation

Consequentially, both *internal and external mapping* were considered during the simulation stage to synchronize events and triggers between the involved tools. All three or just two of the tools can run synchronized this way. Whatever change is occurring in one of the tools, it is transferred to the others so that these can react accordingly. Triggers are sent to the shared communication channel, which was implemented as an event bus. The Kinect gesture recognizer and the *gesture simulator* simply send the *sync triggers* that correspond to the current gesture recognition. In Pidoco each incoming trigger is checked if it is valid for the current page. This is true if a reaction is specified for the trigger. In that case, the reaction is executed in the UI prototype.

For the Yakindu tool the *co-simulator* of our environment uses the mapping table of the *mapping editor* to translate Yakindu events to *sync triggers* and vice versa. This mechanism works continuously while in simulation mode, forwarding the messages between the participating software tools. If an incoming event meets the conditions of a transition as defined in the statechart inside Yakindu the simulation of the statechart continues.

4 Generalizability of the Approach

Our environment is applicable to other combinations of tools, as well as to other interaction techniques, like touch or mouse control. As a prerequisite it must be possible to define a common set of *sync triggers*. Then, the different involved applications can be controlled using either the *internal or the external mapping* and can interact via the event bus, without changes in our environment.

If a tool provides a message interface and the option to retrieve the states (and changes to these states) of the objects of the application under development, the mapping can be done within the external *mapping editor* with minimal changes to the existing code of the tool. In that case, to realize the connection to our environment, the set of relevant events must be exported from the target application as a JSON file. It is then bound to the *sync triggers* inside our *mapping editor*. During runtime the *co-simulator* sends the corresponding events to the tool, which can react accordingly. Only both functionalities, the export and the receiving of external events would need to be added to a tool, in case it does not provide it originally.

If a software tool is open to modifications, the mapping of *sync triggers* to tool functionality may alternatively be implemented within the tool itself. This results in more flexibility to react to specific semantics of gesture bindings within a particular software and is advantageous if for example the UI design options provided by the tool shall specifically reflect the additional concept of gestures.

The mapping concept presented here can be extended to more than the three regarded design spaces (i.e., gestures, presentation and dialog model) and may, for example, additionally include task models or other models to describe the interactive behavior of software applications like the Interaction Flow Modeling Language (IFML)[12]. For that purpose, it may be necessary to extend our *mapping editor* and *co-simulator* to support new types of events.

Even an alternative gesture recognizer can be used, e.g. the Leap Motion sensor[13]. We plan to support it in the future, because of its ability to recognize hand and finger gestures. After the usual binding, as explained above, the Leap gestures would be processed by our environment in the same way as the Kinect gestures from our case study.

[12] http://www.ifml.org/.

[13] https://www.leapmotion.com/.

5 Discussion and Future Work

In this paper, we present our work on an environment for rapid prototyping of 3D gesture interactions. To support the early phases in the design of 3D gesture interactions we consider three design matters: gestures, dialog model and presentation. Rather than developing completely new software to cover the tremendous breadth of required functionality we decided to combine already known and established software tools that developers are familiar with. We show a case study on how to combine Microsoft Kinect gestures with Yakindu statecharts (for dialog modeling) and Pidoco rapid UI prototypes within a novel, integrated environment that serves a coherent design process. It allows detailed consideration of the mutual dependencies of these three design matters and enables designers to change perspectives by switching between them flexibly during the design and evaluation of applications with 3D gesture interactions.

We designed the functionality of the proposed environment based on a thorough user-centered requirements engineering with industrial suppliers of software tools and with partners that hold relevant use cases. An evaluation of the described resulting implementation with real users, to find out how beneficial our approach is for designers, is still under work. Parts of the environment as implemented in the described case study (with Pidoco and the *gesture simulator*) were used by computer science students of our university as part of their study projects building and evaluating gesture interaction applications. We found out, that they could work effectively with the separate tools and the combination of both even without any previous training. Only minor issues concerning the usability of the UI of the environment (e.g. naming of buttons) occurred but were considered for the further development of our environment.

The core parts of the proposed environment are, first, two kinds of mapping as ways to bind gestures to prototypes and to statecharts, and, second, a *co-simulator*. If existing software is open for extension to import sets of gestures and bind them to their control flow, mapping can be done inside the existing tool (*internal mapping*). If changes to existing software must be kept to a minimum, instead an API is required to trigger and export events to bind them to gestures in a separate mapping editor that was developed especially for this purpose (*external mapping*). Both types of mapping are used at runtime to evaluate the gestures in the context of the UI prototype and dialog model respectively. The *co-simulator* guarantees the synchronized execution of prototype and dialog model once a gesture is recognized or its respective event is triggered.

The two proposed approaches for mapping in combination with the implemented event bus for bi-directional communication provide flexibility for the integration of different types of software tools into the environment and ensure high interchangeability of implementations. Alternative tools only need to support one of the two described mappings and then can be combined with our environment.

However, using different software tools can also cause inconsistency in the semantics of the overall application design. With this work a synchronized simulation environment is proposed, but not a synchronized model. This necessitates future work that ensures consistency between presentation and dialog.

In addition, gestures are currently translated simply into sync triggers. But there is more information that could be exchanged meaningfully between applications, like values of variables and attributes from the dialog model to the presentation. For example, counter variables held in Yakindu could be reflected in a Pidoco prototype to enrich the prototyped UI with some functionality provided by the dialog model. Or gestures can consist of several events (cf. the granularity problem in Spano et al. [9]) or must be traced continuously, resulting in more complex *sync triggers*. Our environment will be expanded to support those increased *sync triggers* as well.

Acknowledgments. This research was financially supported by the German Federal Ministry of Education and Research within the framework of the program Forschung an Fachhochschulen - IngenieurNachwuchs (project no. 03FH007IX5).

References

1. Barboni, E., Ladry, J.-F., Navarre, D., Palanque, P., Winckler, M.: Beyond modelling: an integrated environment supporting co-execution of tasks and systems models. In: Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 165–174. ACM, New York (2010)
2. Bomsdorf, B., Blum, R.: Early prototyping of 3D-gesture interaction within the presentation-gesture-dialog design space. In: Kurosu, M. (ed.) HCI 2014. LNCS, vol. 8511, pp. 12–23. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07230-2_2
3. Bomsdorf, B., Blum, R., Hesse, S., Heinz, P.: WeBewIn: Rapid Prototyping bewegungs-basierter Interaktionen. In: Boll, S., Maaß, S., Malaka, R. (eds.) Mensch & Computer 2013: Interaktive Vielfalt, pp. 251–260. Oldenbourg Verlag, München (2013)
4. Feuerstack, S., Anjo, M.D.S., Pizzolato, E.B.: Model-based design and generation of a gesture-based user interface navigation control. In: Gomes, A.S. (ed.) Proceedings of the 10th Brazilian Symposium on Human Factors in Computing Systems and the 5th Latin American Conference on Human-Computer Interaction, pp. 227–231. Brazilian Computer Society, Porto Alegre (2011)
5. Kistner, G., Nuernberger, C.: Developing user interfaces using SCXML statecharts. In: Schnelle-Walka, D., Radomski, S., Lager, T., Barnett, J., Dahl, D., Mühlhäuser, M. (eds.) Proceedings of the 1st EICS Workshop on Engineering Interactive Computer Systems with SCXML, pp. 5–11 (2014)
6. Martinie, C., Navarre, D., Palanque, P., Fayollas, C.: A generic tool-supported framework for coupling task models and interactive applications. In: Ziegler, J. (ed.) Proceedings of 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp. 244–253 (2015)
7. Meixner, G., Paternó, F., Vanderdonckt, J.: Past, Present, and Future of Model-Based User Interface Development, i-com 10, 2–11 (2011)
8. Pintér, G., Micskei, Z.I., Majzik, I.: Supporting design and development of safety critical applications by model based tools. In: Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös nominatae Sectio Computatorica, pp. 61–78. ELTE (2009)
9. Spano, L.D., Cisternino, A., Paternò, F., Fenu, G.: GestIT: a declarative and compositional framework for multiplatform gesture definition. In: Forbrig, P., Dewan, P., Harrison, M., Luyten, K. (eds.) Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS 13, p. 187. ACM Press, New York (2013)
10. van Buskirk, R., Moroney, B.W.: Extending prototyping. IBM Syst. J. **42**, 613–623 (2003)
11. van den Bergh, J., Sahni, D., Haesen, M., Luyten, K., Coninx, K.: GRIP. In: Paternò, F., Luyten, K., Maurer, F. (eds.) Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems - EICS 2011, p. 143. ACM Press, New York (2011)