Electronic version of an article published in International Journal of Cooperative Information Systems, Volume 20, Issue 2, 2011, Pages 177-200, DOI http://dx.doi.org/10.1142/S0218843011002195

UNIFIED MODELING OF SERVICE LOGIC WITH USER INTERFACES

FRANK ALEXANDER KRAEMER
SURYA BAHADUR KATHAYAT
ROLV BRÆK

Department of Telematics, Norwegian University of Science and Technology (NTNU)
O.S. Bragstads plass 2a, 7491 Trondheim, Norway
{*kraemer, surya, rolv.braek*}*@item.ntnu.no*

We describe a method based on UML activities for the unified specification of collaborative service behavior and local user interfaces. The method enables a model-driven development process, which effectively combines the need to express service collaborations involving several components with the need to provide detailed operations for user interfaces. Our service models use activities as the primary building blocks that encapsulate self-contained functionalities. We show, how a complete distributed system can be decomposed into such building blocks, how this decomposition leads to a separation of user interface concerns from service collaboration concerns, and how they may be combined with an event-driven composition mechanism based on activity parameter nodes. We also demonstrate how different UI frameworks can be supported, and illustrate the method with a case study of a situated collaborative learning service.

Keywords: Model-Driven Development; Service-Oriented Architecture; User Interfaces; UML Activities; UML Collaborations.

1. Introduction

During the development of mobile services for end-users, a major challenge is the difference of perspective that some parts of the system demand. On the one hand, cross-cutting service behaviors need to be specified so that each component of the system may interact consistently with the other parts. A challenge here is the complexity that arises from the coordination of distributed behavior in general and asynchronous, multi-initiative peer-to-peer interaction in particular. On the other hand, end-users expect highly sophisticated and responsive user interfaces. Not only do these depend on the availability of certain libraries for user interfaces, like for instance Java Swing or Android, but they should also fit perfectly with the device they are executed on, to match screen size and input methods, for example. This implies a level of details that, using conventional modeling and programming techniques, is hard to combine with the cross-cutting view on service interactions needed to get the overall system right.

What unites these two perspective is their reactive nature; both user interfaces as well as distributed communication must react on events from an environment, that means user input or the reception of signals. Furthermore, they are tightly coupled by such events: user inputs may trigger communication with other devices, and, vice versa, the arrival of signals triggers changes in the user interface.

Our method, first described in [1] and complemented in [2], therefore focuses on the specification of reactive behavior, how it may be encapsulated in the form of special building blocks, and how it may be composed effectively to achieve the desired overall system functionality. In this paper, we focus on the combination of collaborative behavior with local user interfaces, and introduce a separation between two kinds of building blocks:

- **Collaborative Service Blocks** model cross-cutting behavior among several components. The major concern with these building blocks is the specification of the coordination necessary to accomplish distributed tasks and communicate data.
- **User Interface Blocks** are executed locally on a device, and encapsulate all interactions from and to the users, as well as the detailed operations on user interface elements, like windows or buttons.

Both kinds of building blocks are equipped with parameter nodes that can be used to compose them together, either to pass data or to notify a block about an event detected by another block. Due to this composition style, the construction of system specifications resembles that of wiring together blocks. Further, we observed that the data types and events needed for composing blocks together, usually correspond to concepts from the problem domain, which makes specifications easier to understand by application developers familiar with a certain domain. The encapsulation of building blocks makes it possible that applications refer to abstract tasks (for example to provide user credentials), and that dedicated implementations may realize this task in the best way for the corresponding platform. To exemplify this, we show two implementations of a UI building block realized for both Java Swing and Android. For the latter we detail a step-by-step method that integrates with the layout editor for user interfaces of the Android SDK.

The novelty of our approach is the way in which collaborative service logic is represented together with the local user interfaces: on the one hand, these concerns are separated into different specification units (the blocks), but on the other hand their events can be composed on the necessary level of detail to build responsive applications. Moreover, the use of building blocks leads to an incremental working process, since they can be developed and analyzed in isolation, and serve as interfaces between experts of different domains. These benefits are further discussed in Sect. 10. The rest of the paper is organized as follows: We describe the system that is subject of our case study in Sect. 2 and continue with a brief introduction to our engineering method in Sect. 3. Then, we describe how the system can be decomposed into subordinate building blocks in Sect. 4. The modeling of collaborative

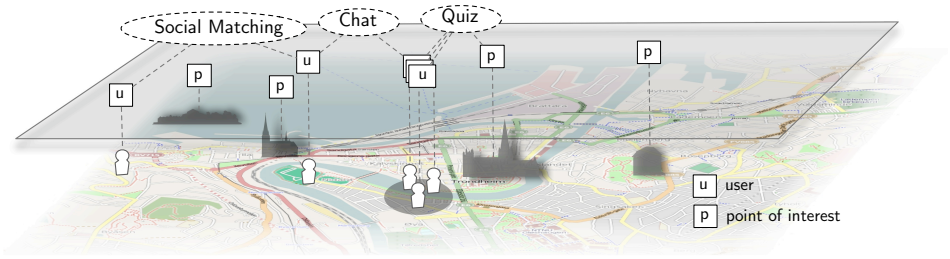

Fig. 1. Mapping of domain objects to system entities and collaborations

services by means of UML collaborations and activities is presented in Sect. 5, followed by the encapsulation of user interfaces in UML activities explained in Sect. 6. After that, Sect. 7 and 8 outline the automated verification and implementation of the specifications. In Sect. 9, we summarize and discuss related approaches and close in Sect. 10 with concluding remarks.

2. Case Study: Exploring the City

As an example, we specify a service for students to learn about different historical places in a city using mobile devices. It is based on the situated learning service introduced in [3]. The system tracks the position of the students and responds when students with similar interests are closely located. Based on this feedback, users may create groups to cooperate. When a user comes close to a point of interest, a location-specific service can be started. Such a service can simply provide information and also engage students in learning activities. For our example, we realized an interactive quiz. During a quiz session, users are asked questions related to the particular point of interest. Users can interact with each other to answer the questions.

As illustrated in Fig. 1, the system is composed of objects reflecting domain entities, such as users, user groups, and different types of point of interests like museums. Between these objects, we identified the following services:

- A social matching service helps to create groups based on user interests.
- A position update service tracks the location of a user and makes it available to other users, for instance other members of the same group.
- A chat service allows the users to interact.
- A number of learning services associated with a particular point of interest are used to support the actual learning, in our case the quiz service.

Figure 2 shows two of the user interfaces involved in the realized system. For our application, we have built user interfaces in Java Swing for laptop computers, as well as user interfaces running on Android for mobile devices. There are considerable differences between these user interfaces, but the service logic is the same. Therefore

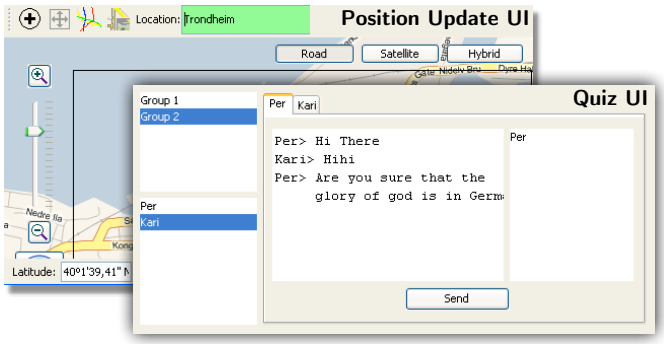

Fig. 2. User interfaces for the individual services

it is desirable to separate these two concerns of the system such that they may later be composed. In the following sections, we describe our service engineering approach that enables a controlled composition of these concerns.

3. The SPACE Engineering Method

The main specification units in our engineering method are special building blocks that are expressed as UML models combined with Java code covering the details of operations. Building blocks may describe the local behavior executed by only one component, but they may also span across *several components* and describe *collaborations* among them. This flexibility is the key to the approach presented in this paper, since we will later use building blocks to encapsulate both local user interface behavior (as single-component blocks) as well as the collaborative service behavior involving several components (as collaborative blocks). Our building blocks have previously proven to enable a high degree of reuse (see [4] for a survey). Therefore, the development of a system starts with the consideration of libraries of reusable

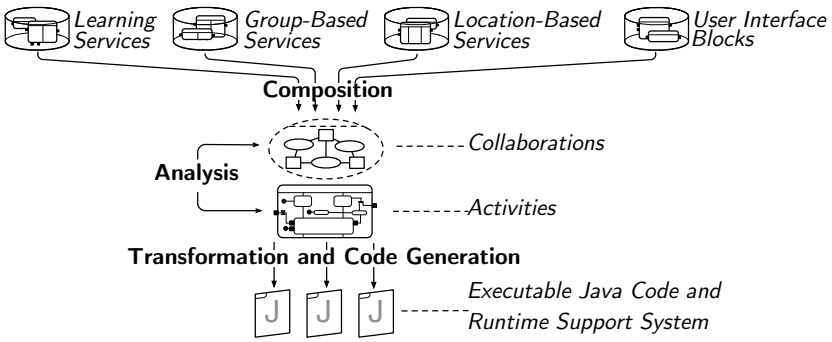

Fig. 3. The SPACE engineering method

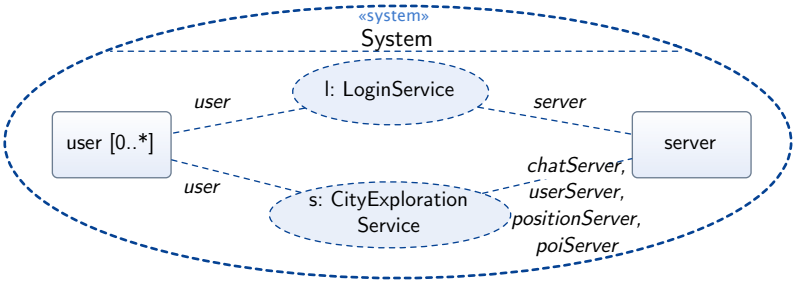

Fig. 4. Collaboration for the complete system

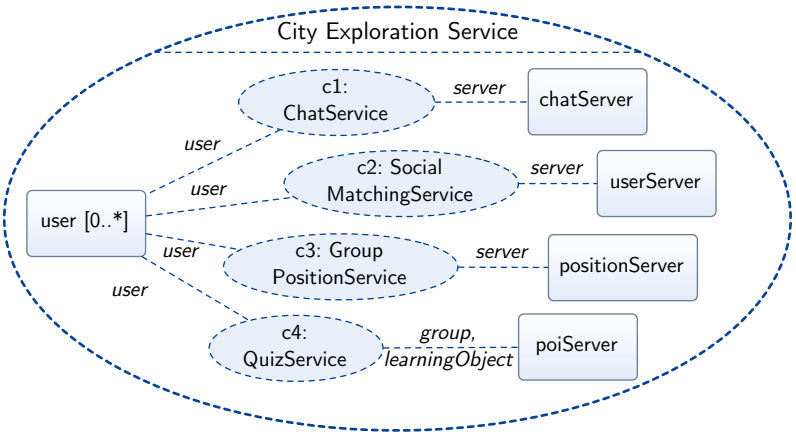

Fig. 5. City Exploration Service Collaboration

building blocks for different domains, as illustrated in Fig. 3. This leads to a drag-and-drop-like specification style, in which building blocks are composed to more comprehensive ones, until a complete system specification is obtained. Functionality not yet available is provided by new building blocks that may be stored for later reuse. Due to the formal semantics underlying our specification style, building blocks can be analyzed by means of model checking, which we explain further in Sect. 7. Once a system specification is complete and sound, it may be implemented in an automated process detailed in Sect. 8.

To capture behavior involving several components, we use *collaborations* as major specification units. Fig. 4 shows the city exploration system on its highest decomposition level in the form of a UML collaboration. It consists of a number of users connected to a central server, represented by the rectangular collaboration roles. The ellipses between them refer to subordinate collaboration uses, namely l to a login service and s to the actual city exploration service. The latter is defined as a composition of collaboration uses, as shown in Fig. 5. The server is partitioned

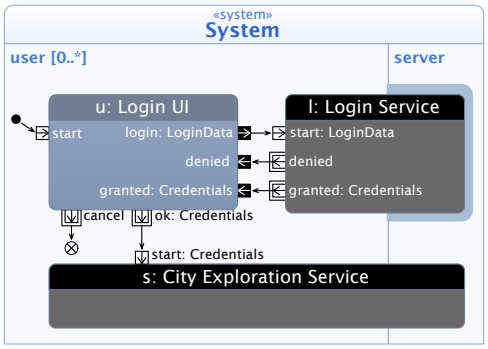

Fig. 6. System activity

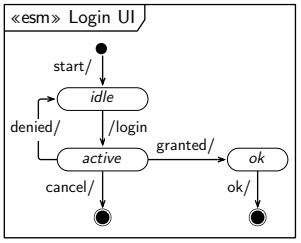

Fig. 7. ESM of the Login UI block

into four collaboration roles: a separate chat server, a user server, a position server and a point-of-interest server. The interactions of these servers with the users are in turn described by collaboration uses *c1...c4* for chat, social matching, group positioning and the quiz service. The labels on the lines that connect collaboration uses to collaboration roles are role bindings.

While the UML collaborations in Fig. 4 and 5 already document some aspects of the system structure and the distribution of responsibilities among the components, they do not specify any detailed behavior. The behavior of each UML collaboration is defined by a corresponding UML activity, shown in Fig. 6 for *System*. The activity contains one partition for each collaboration role, (i.e., *user* and *server* for the top-level system), which are usually assigned to different executable components. Each subordinate collaboration is represented by a call behavior action. For example, the login service *l* and the city exploration service *s* in Fig. 4 are represented by the call behavior actions *l* and *s* in Fig. 6. Note that they cross the partition borders since they are executed by users as well as a server. Lighter shaded blocks, such as *u: Login UI* refer to local building blocks, which are only represented in the activity perspective. Building blocks defined as UML activities have pins labeled by detailed events. These refer to the externally visible events that can be used to link blocks together, possibly inserting some control logic between them. The additional shadow at the server side of the login service indicates that the server handles several instances of them, as we detail later.

Users start their services by activating the user interface for the login, represented by block *u* that is started via initial pin *start* following the initial node. On this level, we are not interested in the internal details of the login user interface. Instead, it is sufficient to consider the external view provided by the special state machine in Fig. 7. It is a so-called external state machine (ESM), that defines the allowed sequences of tokens passing through pins. From this ESM we read that, after the login block has been started, it will emit a token via pin *login*. This pin is drawn filled, denoting that it is a streaming pin, which can pass tokens while an

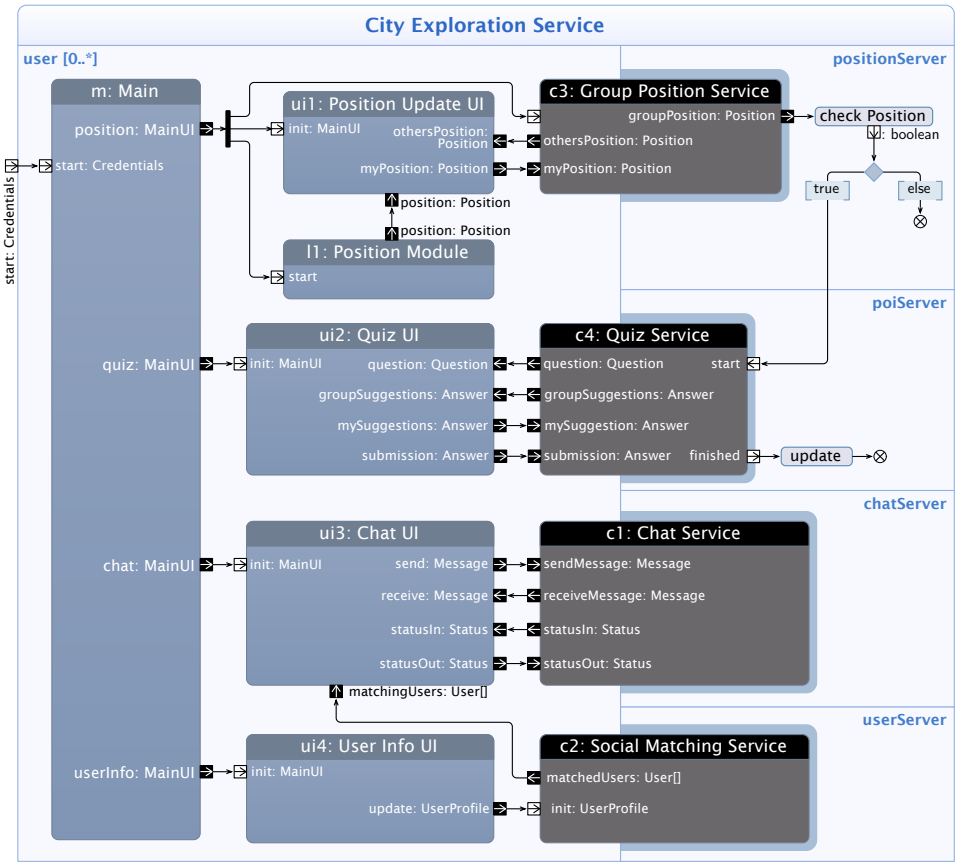

Fig. 8. Activity for the City Exploration Service

activity is active. The provided login data is used to start the actual login service as a collaboration with the server. As shown in Fig. 6, this collaborative building block has no streaming pins, and may only terminate in two alternative ways, represented by the two pins *granted* and *denied*. (These are mutually exclusive and belong to different parameter sets, noted by the additional box around them.) The corresponding result is fed back to the login user interface block, which will either forward a successful login via *ok* or eventually cancel. In the latter case, the flow simply ends in the flow final node. In case of success, the actual city exploration service is started via its pin *start*.

The decomposition of the login phase into a *Login UI* and a *Login Service* from Fig. 6 is a typical example for the general pattern that we ise for the separation of service logic from user interface logic. The local block *Login UI* encapsulates all logic specific for user interfaces, and the *Login Service* block models the collaborative behavior. We explain the internals of these blocks in Sect. 5 and 6.

4. System Decomposition

The decomposition of the city exploration service into its sub-services was already introduced in Fig. 5 with respect to the collaborations between the entities; Fig. 8 shows the corresponding decomposition in the activity diagram view, which also adds details about the coupling of events between the sub-services, and, as the main focus in this paper, the detailed user interface blocks. Each user interface part as the ones illustrated in Fig. 2 is represented by a separate UI block in the activity for the city exploration. In the Java Swing UI framework, these correspond to separate windows. For Android applications running on mobile phones, these correspond to different application screens, as we shall later see.

When a user successfully logs into the system, a user component of the city exploration service is started via starting node *start*, which activates the life-cycle controller of the city explore service *m: Main*, from which the other UI elements are controlled. At the beginning, the position module *l1* and the UI block for the position update UI are activated. The position module is continuously forwarding the position of the user to the UI via pin *position*. The group positioning service receives position updates from every user and forwards them to the other members of the group via the *othersPosition* streaming output pin. The Position Update UI then refreshes the position information of the user and other group members on the map.

The position server checks with operation *checkPosition* if any member of a group is close to a point of interest. In such a case, it informs the point of interest server to start the quiz service. The Quiz service then forwards the question via its streaming outpin *question* to all members of the group (detailed in Sect. 5). During the quiz service, all the user can suggest an answer via *mySuggestion* streaming output pin of the *quiz UI* block which is connected to the streaming input pin *mySuggestion* of the quiz service. This suggestion is then forwarded to the other members of the group and group suggestions are updated to the quiz UI. One group member is assigned to be the group leader and submits the final answer via output pin *submission* of the quiz UI.

In parallel with the quiz and positioning services runs the social matching service. Users specify their interests and create profiles via *User Info UI*. The social matching service then returns the list of matched users through its output streaming pin *matchingUsers*. A list of matched users is also made available to the chat user interface component *Chat UI*, so that users can communicate with each other using chat service *c1*.

5. Collaborative Service Blocks

Fig. 9 shows the activity for the Login Service. It is activated by providing the login credentials via pin *start*, which are forwarded to the server role. There, the login data is checked, whereupon the service eventually terminates at the client side by either *granted* or *denied*.

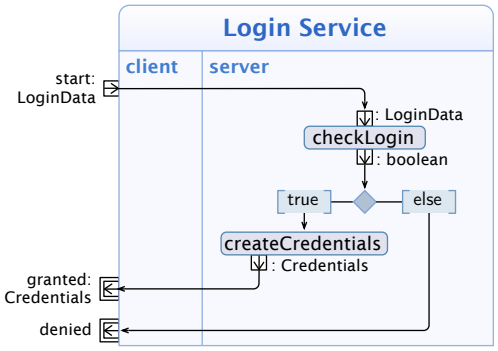

Fig. 9. Activity for the Login Service

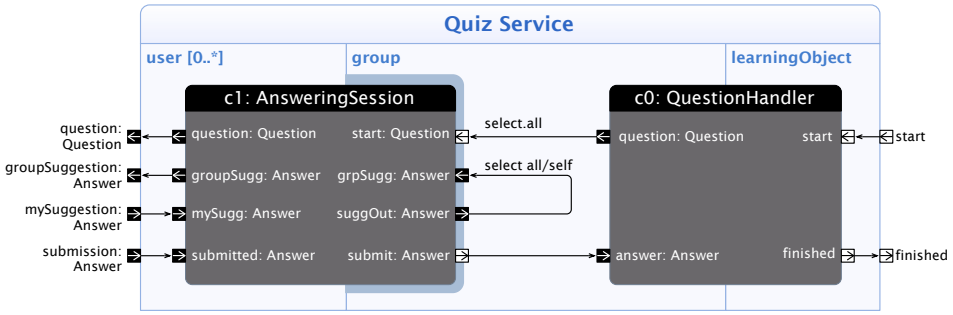

Fig. 10. Quiz Service

The Quiz Service is shown in Fig. 10. Within one execution of the service, one learning object and one group object participate, as well as any number of users in a group, here emphasized by the multiplicity [0..∗]. Note that due to the role binding in Fig. 5, the groups and learning objects are provided by the partition *poiServer* in Fig. 8. The question handler service *c0* is responsible for providing the questions and evaluating the answers. Once a question is provided, it is sent to the answering session service. As indicated by the shadow, this collaboration is executed separately for each participating user and the group therefore handles multiple instances of it. The question is provided to *all* users, declared by the operator **select all**. The question is then forwarded within the answering session. Suggestions by the users are picked up via pin *mySugg*, internally forwarded to pin *suggOut* within the group partition. From there, they are distributed to all *other* users, declared by operator **select all/self**. These selection operators work as symbolic address filters and are further explained in [5].

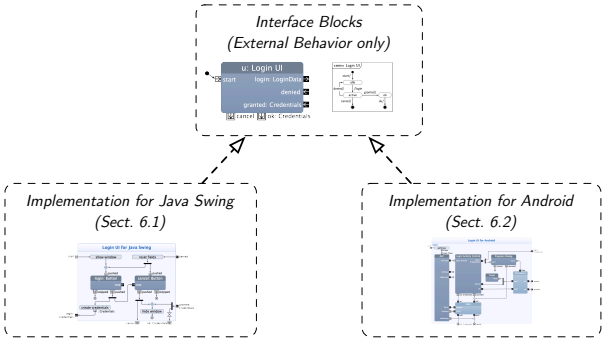

Fig. 11. Realizations of *Login UI* for Swing and Android

6. User Interface Blocks

Just as the behavior of the distributed services, the behavior of user interfaces is triggered by distinct events. In addition to the triggers from signal receptions and timeouts, also direct actions from users (like the tap on a button) must be taken into account. In contrast to the distributed service behavior, however, the *internal* behavior for user interfaces is highly dependent on the specific devices they are executed on: not only do different platforms (like Java Swing or Android) offer different user interface elements and layouts, they also assume different life cycles of these elements which has influence on their overall behavior. Interestingly, we observe that the *external* events often are the same so that these differences may be encapsulated. User interface blocks may therefore have the same pins and the same ESMs while having very different internal realizations. For this reason, user interfaces may in the first place be described by an abstract block that only defines their external behavior such as the one from Fig. 7. Other building blocks may implement this external behavior, expressed in UML as the *realization* dependencies depicted in Fig. 11. This means that an application can be ported to different devices by exchanging only the internals of the specific building blocks, while the overall composition and especially the coupling to the service logic can stay unchanged. Our transformation tool therefore selects implementations of the Login UI block depending on the desired target platform. (The actual selection mechanism is part of the deployment and not detailed here.) Since the ESMs are identical, the overall applications will behave equivalent.

In the following, we will start in Sect. 6.1 by presenting one such implementation of *Login UI* with a simple building block for Java Swing, which introduces the mechanisms for the coupling to user interface elements. In Sect. 6.2 we extend our method and design another implementation of *Login UI*, specific for Android, which makes use of the layout editor of the Android SDK and takes care of the particularities of mobile devices.

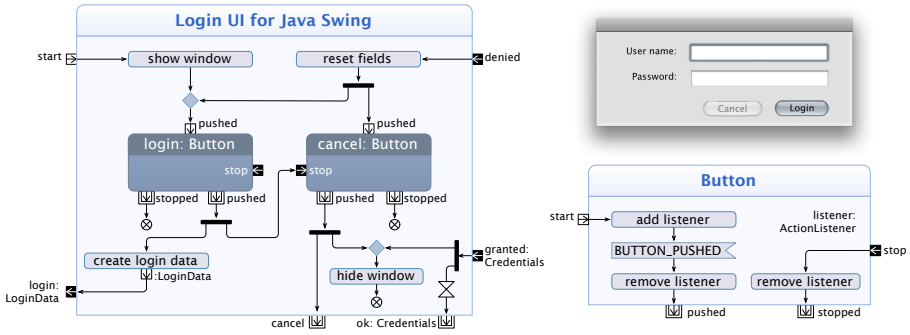

Fig. 12. Activities for *Login UI* and *Button*

6.1. *Simple User Interface Block for Java Swing*

Figure 12 shows the Login UI building block specific to Java Swing. It contains two buttons, one for sending the login data and one for canceling, each represented by a corresponding building block. The internals of the buttons are presented to the right. Once a button is started, it registers a listener to the graphical element. When this listener is activated, it sends an internal signal *BUTTON_PUSHED* to the underlying runtime scheduler. The behavior triggered by this signal follows after the accept signal action for *BUTTON_PUSHED*: the listener is removed and the button block terminates via *pushed*. To deactivate a button, a token may be sent via *stop*, upon which the block is terminated. The text field is created together with the window and the other elements, but since it does not trigger any events, no further elements are necessary for it in the model.

The external behavior of *Login UI* is defined by the ESM given in Fig. 7. After its start, it shows the window illustrated at the right hand side of Fig. 12, with the login button activated. When the user presses login, a token leaves block *login* via *pushed*, whereupon the login data is created from the user name and password provided and sent out. From then on, the block awaits the arrival of either *denied* or *granted*. In the denied case, users may select to retry or to cancel using the then activated cancel button. In the granted case, the *Login UI* block terminates via *ok*.[a]

A similar approach works for all the other user interface blocks as well, for instance the *Quiz UI*. Its external behavior is shown in Fig. 13. Once it is started, it goes to the *idle* state and changes into *active* once it receives a question. In this state, users may suggest the answer to the question while they receive suggestions from the group members. Once the final answer is submitted, the quiz UI enters into the *submitted* state and waits for the result. The internals of the block are similar to the ones from the *Login UI* in Fig. 12. Operations are used to update the

[a]The ESM from Fig. 7 demands that parameters *granted* and *ok* are part of separate execution steps, for instance to leave time for releasing resources. Since the operation to hide a window in Fig. 12 is executed within one step, an intermediate delay element is added.

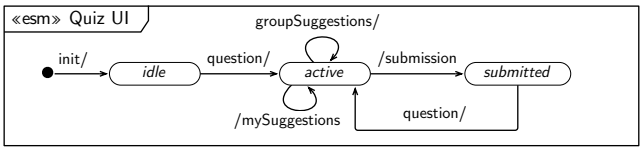

Fig. 13. ESM of the QuizUI building block

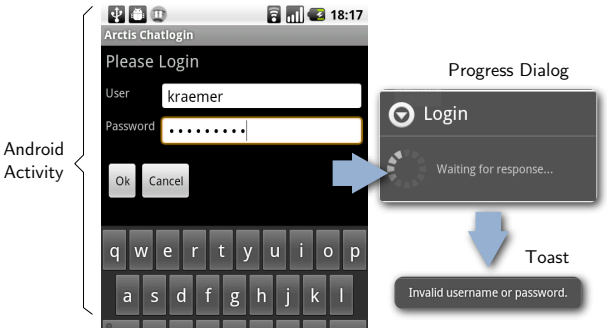

Fig. 14. Screenshot of the login user interface on Android

state of the UI elements and register listeners to them, which send events back to the building block.

6.2. *User Interface Blocks for Android*

Figure 14 shows the login user interface on Android. Screen layouts as the one illustrated are called *activities*. (To distinguish them from UML activities, we always refer to them as *Android activities* in the following.) In addition, a progress dialog is shown during the login, and the result is displayed with an Android-specific user notification, called *toast*. While the different user interface elements may look similar to those for Java Swing, there are some important differences:

- Only a single Android activity can be displayed on the screen at once. This means in particular that displaying several windows as on desktops is not possible.
- Since applications may be interrupted by other tasks, for instance an incoming call, any Android activity must be prepared to be moved into the background. In case the memory gets low, Android activities also have to be prepared to store their state and wait for a later re-activation.
- The appearance of user interfaces is determined by rather general layout files, which are created by graphical tools like the ones provided by the Android SDK. The instantiation of the user interfaces in terms of an object structure is done by the operating system based on this file, taking

into account the current device configuration, regarding screen size and orientation, for instance.

For a modeling approach, this imposes several challenges, since the lifecycle of Android activities has to be taken into account, and the layout files somehow have to be connected to the building blocks in UML. To meet these additional challenges, we use the method depicted in Fig. 15, which first produces a building block to encapsulate the Android activity and then composes this block further.

(1) The layout file is created with a graphical editor. In the example, this file arranges the text fields and buttons shown in Fig. 14.
(2) Events are identified that originate at the user interface and that trigger service logic (such as the activation of the button *OK*). Vice-versa, events are identified that update the user interface, such as the arrival of a login denial.
(3) With an ESM, constrains on the sequence of the events identified above are described. In the example, the ESM is similar to the one in Fig. 7.

The result of these steps is the external shell of a *layout control block* that encapsulates the user interface elements of the Android activity, in our example called *Login Activity Control*. Since all steps are focused on the visible part of user interfaces and in which sequence users should interact with them, they can be accomplished by a user interface designer without specific programming skills. In the following, a programmer adds the internals to this block and composes it with other blocks to form the final UI block:

(4) The internals are added to the layout control block. These are methods and listeners that interact with the elements defined in the layout file, similar to how the listeners and operations work in the block of Fig. 12.
(5) The contents of the methods is edited to update the user interface elements upon events (for incoming events) and listeners are registered to catch events originating at the user interface.
(6) In a final step, the layout control block is combined with other user interface elements such as dialogs from a UI library to create a comprehensive Android UI block. A special block taking care of the lifecycle of an Android activity is added as well.

As a result of this method, we obtain the Login UI block for Android shown in Fig. 16. From an external view, it behaves as described by *Login UI* from Fig. 7. Internally, it is composed from several blocks and adheres to Android's particularities:

- Block *ALC* (for *Android Lifecycle Controller*) creates the Android activity for the login screen. Since the creation must be done by the operating system, only the class is passed to it. The instance is returned via the pin *onCreate*. Furthermore, *ALC* monitors the lifecycle of the activity. If another application is coming to the foreground, the pins *onPause* and *onResume* trigger an event

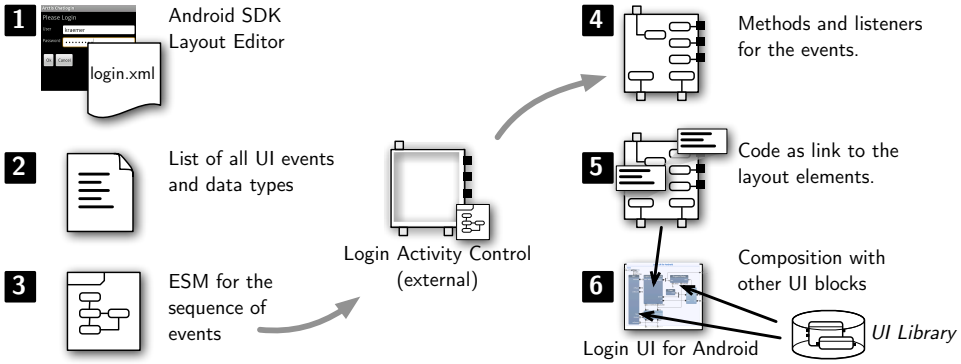

Fig. 15. Method to create Android UI Blocks

upon which data can be persisted. (Not shown here.) Eventually, the Android activity can be terminated via *finish*, or is destroyed by the operating system via *onDestroy*.

- Block *Login Activity Control* makes the user interface elements of the Android activity available, as described above. Input pins update the text fields and output pins forward events that originate from the buttons.
- Block *Progress Dialog* displays a dialog while the login credentials are evaluated by the server. The user may also cancel the login process using this dialog, which is expressed by the corresponding pin.
- Block *Toast* displays a message in case the login was denied.

The blocks *Gate* and *Crossover* are taken from our standard library and are used to intercept two alternative flows that arrive to them via *in1* and *in2*. *Crossover* terminates the progress dialog upon the arrival of either *denied* or *ok* via *flow*, and *Gate* forwards *ok* or *cancel* to the termination but intercepts this forwarding by a termination of the Android activity via block *ALC*.

7. Validation and Automatic Verification

Due to the formal semantics [6], the specifications expressed by the UML activities can be analyzed by the model checking tools described in [7, 8]. The encapsulation of building blocks in their ESMs leads to a compositional verification style, in which each building block can be analyzed separately. When a specification is composed of several blocks, for instance several services and UI blocks as in Fig. 8, then all its subordinate blocks are abstracted by their ESMs. This keeps the state space of the analysis rather small, as shown in [7]. In addition, it is possible to study design solutions that are not completely finished yet. For instance, the block for the login on Android in Fig. 16 can be validated even if the internals of *Login UI Control* are not yet finished. This can provide feedback at earlier stages of the development, which reduces costs for changes.

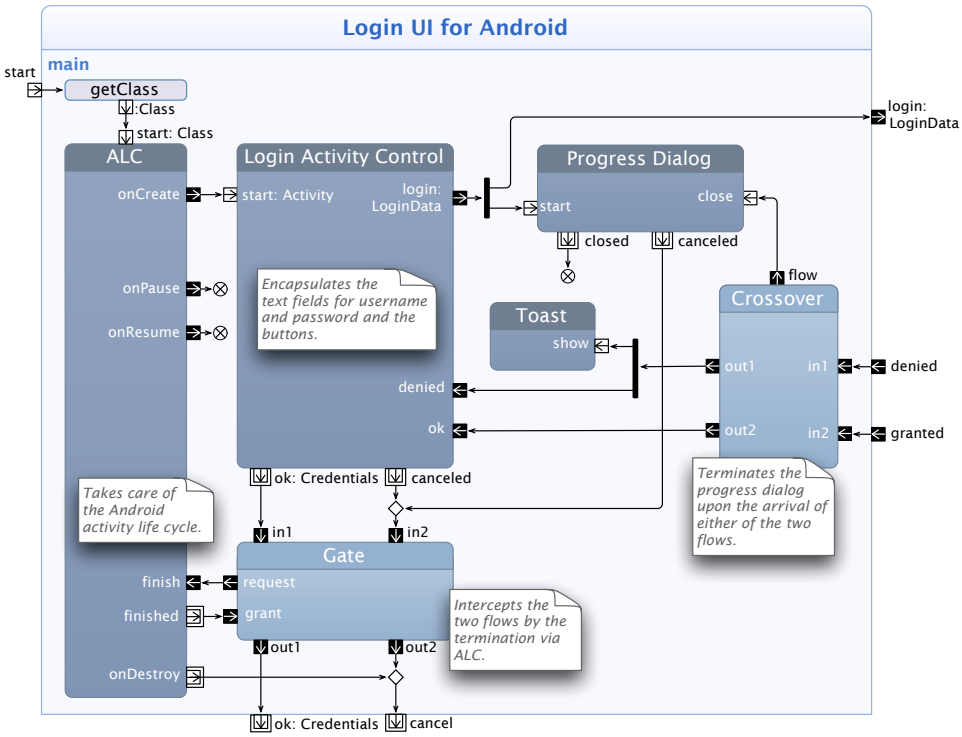

Fig. 16. Android version for the login block

As a means of validation, i.e., whether a behavior is suitable to solve a certain problem, the behavior of a building block may be simulated by means of a graphical animation, as shown in [8]. In such a simulation, designers can step through possible behaviors by looking at the sequences of actions and states. The actions are triggered by events and the states are defined by the ESM states of the inner blocks and other elements of the UML activities.

In addition to the validation that relies on the judgement of the designer examining selected paths through the state space implied by a specification, a thorough and automatic verification is possible that takes the entire state space into account. Such an analysis reveals errors in the interactions within collaborations, for example unbounded queues, deadlocks, race conditions and inconsistent terminations, i.e., situations that are generally undesired and that are most likely design flaws. (This means the given specification is verified against a set of desirable properties.) With respect to the user interface blocks proposed here, two properties deserve special attention:

- A building block must obey its own ESM description. For instance, the *Login UI* block may push a token through *ok* if (and only if) it received a *granted*,

since its ESM allows these events only in that order.
- A composition of building blocks must obey the ESMs of all the blocks. For instance, the login gui block may only be composed in such a way that, once it has emitted a *login*, it will eventually receive either a *denied* or *granted* (but not both of them). This is very useful for user interfaces that maintain a certain state (for example if a UI element is enabled or not) and relieves the developer of ensuring these conditions manually.

Once errors are detected, the model is annotated and may be animated to help the designer to understand the error situation and correct it, as shown in [8].

8. Automatic Implementation

To implement the specifications given in terms of UML activities and the complementing Java methods for the call operation actions, we developed a two-step process. In a first step, the activities are transformed into UML state machines. Each partition of the system activity in Fig. 6 denotes a separate component, for which we generate a UML class. The transformation considers which collaboration roles are bound to the respective components and generates state machines for the respective activity partitions. This transformation is detailed in [2].

In a second step, code is generated from the UML state machines. The state machine logic is translated to special transition methods that execute the state machine transition actions, such as sending signals to other machines, setting timers or executing the Java operations that have been copied directly from the building blocks. Our code generators produce code for different platforms, such as standard Java [9], embedded Sun SPOT devices [10], as well as Android [11].

9. Related Work

In general, we observe that some service engineering approaches ignore the details of user interfaces completely and leave such aspects to the implementations. Other approaches have user interfaces as their primary focus, but treat the service logic as secondary.

The possibilities of model-driven user-interface development have been explored for instance in [12–14]. Most of these approaches focus on web applications. In [12], user interface behavior is modeled with UML use case diagrams which are detailed with activity diagrams showing the interactions between users and the system. User interface components such as Java applets are then generated. Link et. al [13] use extended UML activity diagrams to capture UI aspects both from the user and the system perspective. UI models are used to specify the assembly of user interfaces components, from which code can be generated. In [14], user interfaces are modeled using several types of UML diagrams. Class diagrams are used for the domain model representing domain entities and their relationships, while activity diagrams are used for task modeling. In contrast to these approaches, we do not

try to automate UI development as such, but provide a way to factor out and encapsulate UI elements from services and use them in application composition at the modeling level.

UWE [15, 16], WebML [17], OO-H [18], and MIDAS [19] are model-driven approaches for the domain of web-based systems. In [15], user interfaces are modeled and implemented by code generation. This approach considers only web services which are mapped to Java methods. UI components just invoke a web service and wait for the result in order to present it. In UWE, information aspects are specified by content models using UML class diagrams. Nodes and links of the hypertext structure are specified in a navigation model using UML class diagrams. Composition of the presentation elements are specified in a presentation model using stereotyped UML class and interaction diagrams. Behavior is specified by the process flow model using UML activity diagrams. Similar to UWE, WebML also has the concepts of structural model for data modeling, composition model for the page contents, and personalization model for the customizing features. All the concepts of WebML are associated with a graphic notation and WebML specifications can also be translated into web pages. In OO-H, structural domain information is captured using UML class diagrams. From there, different navigation models are created for each user type. Then, using different mapping steps, a default web-interface is generated. Presentation models based on templates are combined with the aid of a set of patterns to improve the quality of the generated interfaces. A model compiler is used to generate the user-interfaces for internet applications. The MIDAS approach has a system core that defines domain and business models. Over this central core, it defines structural and behavioral dimensions of the web-application using conceptual and platform-specific models for content, hypertext and presentation. A common and interesting feature of these methods is the modeling of the application in different orthogonal levels and aspects such as content, hypertext, composition and presentation modeling. One limitation of these approaches is that they consider only web applications and client-server type of services. In contrast, we provide an approach for reactive services in general without technology bindings. Our service models are collaborative and encapsulate the details of interactions and distribution. We also treat UI elements in the same way as service behavior.

SoaML [20] is a UML profile for the structural aspects of the service. Services contracts are modeled in a way similar to your approach of specifying collaborative services. UML4SOA [21] is a profile for specifying behavioral aspects of services. It contains specialized elements for modeling service interactions, compensation, event and exception handling. Neither of these UML profiles, however, deal with details of user-interfaces.

Since UML in general does not provide any dedicated concepts or diagrams for developing user interfaces, some approaches use profiles specifically for user interface design. For instance, [22] presents how UML activity diagrams can be used to show the flow of windows and other UI elements, by representing actions that a user can invoke while working with the user interface. The approach introduces stereotypes

to further categorize these actions. In comparison to our work the focus lies on the navigation between local windows, but no connection to service logic or its coupling with events. Van den Berg and Coninx [23] propose a UML profile for the description of user interfaces in relation to context models that represent the situations and environment in which the interface is used. Similar to our approach is the use of UML activity diagrams, but the emphasis on what they represent differs. Our models are focused on the technicalities of user interfaces and are very detailed with respect to the event-driven behavior, to an extend that allows code generation from these models. In contrast, their models pay more attention to the relations between the system, the user and the environment by representing the latter explicitly in the diagrams.

App Inventor [24] is a visual programming environment for creating mobile applications by connecting visual boxes like puzzle pieces. Instead of writing code, the programmer specifies application logic using boxes for specific functionalities along with control constructs that realize programming structures like conditions and loops. Its development environment is similar to StarLogo TNG [25], and Lego Mindstorms [26] but targeting mobile applications for Android. The diagrams of App Inventor resemble Nassi-Shneiderman diagrams known from structured programming, while the activity diagrams underlying our approach model general data flows, which also offer synchronizing elements like join nodes. It is therefore not clear to us how more general reactive behaviors, in which reactions on events depend on complex states, can be expressed, especially if more than one event need to be synchronized. In our approach, the internal details of the services and UI blocks are encapsulated with their external behavior specified using ESM. This allows replacement of internal details while keeping the external behavior. Moreover, our building blocks are composed together by using different types of pins such as initiating, terminating and streaming pins linked together by arbitrary synchronizing logic. We also separate user interface and service concerns but compose in a unified way such that their composition can be verified and validated (automatically) early at the design time. As in our approach, the layouts for graphical user interfaces need to be designed by hand in App Inventor, and their association with corresponding functionality blocks is done manually.

10. Concluding Remarks

We proposed a method to integrate collaborative service behavior with the local control of user interfaces using building blocks based on UML activities. We observe that the demonstrated specification style leads to a system decomposition in which user interface elements are represented by separate, self-contained building blocks that may be developed by UI experts. These may be combined with collaborative service blocks to form the complete system specification, which can then be analyzed and implemented in automated processes. Since the proposed specification style is an extension of our existing method for reactive system engineering, it inherits

several properties that we consider as beneficial for the development of services in general:

- The decomposition into collaborative building blocks based on activities models systems in reasonably compact but readable form. Our case study involves multiple users and mobile devices communicating with each other using a variety of services each involving several system participants. The complexity of the system therefore goes beyond that of simple toy examples. However, we are still able to present its overall specifications on a few pages, as shown by Fig. 6 and 8.
- Interactions and coordination of collaborative behavior is handled explicitly on the service model level, with activities that provide an overview of the cross-cutting behavior executed by several components.
- The automated and incremental verification encourages developers to formally analyze their specifications often and from the beginning, block by block.
- The automated implementation makes the service specifications the canonical description from which everything else is derived. This avoids inconsistencies.

With respect to the integration of user interface behavior, our approach has several important properties:

- **Encapsulation of UI details.** The building blocks encapsulate detailed operations on user interfaces, which would otherwise obstruct the overall specification and which would make it difficult to understand the cross-cutting services.
- **Separation of expertise**. Due to the separation of concerns enabled by the decomposition into building blocks, developers with different fields of expertise may work largely independently from each other.
- **Protection of operation call sequences.** The above mentioned separation could also be achieved by object-oriented techniques with classes separated by interfaces. But by using building blocks, we also ensure that operations are called in the right order, secured by the ESMs. Furthermore, since building blocks are at the service model level, the analysis of the overall behavior is easier than at the code level.
- **Application-Oriented Composition.** The coupling by means of activity flows is application-oriented: data types and events correspond to concepts of the domain, like the types *Position* and *User* in Fig. 8. This makes specifications easier to understand.
- **Interchangeability of UI Frameworks.** Since operations and resources belonging to a certain UI are encapsulated as building blocks, they can be exchanged easily so that a system may use different UI frameworks. If the new building blocks adhere to the same ESMs, they will behave consistently.

So far, we encapsulate user interfaces into building blocks in a manual, although highly structured method as outlined in Fig. 15. Since this invokes recurring patterns, we see potential for the automation of this process and think of solutions

that encapsulates artifacts produced by GUI builders for different UI frameworks automatically. Especially the creation of building blocks controlling the Android activities can be automated further, taking the layout files created by the Android SDK as input.

References

1. Frank Alexander Kraemer and Peter Herrmann. Service Specification by Composition of Collaborations — An Example. In *Proceedings of the 2006 WI-IAT Workshops (2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology)*, pp. 129–133. IEEE Computer Society, 2006. 2nd International Workshop on Service Composition (Sercomp), Hong Kong.
2. Frank Alexander Kraemer. *Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks*. PhD thesis, Norwegian University of Science and Technology, August 2008.
3. Surya Bahadur Kathayat and Rolv Bræk. Platform Support for Situated Collaborative Learning. In *International Conference on Mobile, Hybrid, and On-line Learning, 2009. ELML '09.*, 2009.
4. F. A. Kraemer and P. Herrmann. Automated Encapsulation of UML Activities for Incremental Development and Verification. In A. Schürr and B. Selic, editors, *Proceedings of the 12th Int. Conference on Model Driven Engineering, Languages and Systems (Models), Denver, Colorado, USA, October 4-9, 2009*, volume 5795 of *Lecture Notes in Computer Science*, pages 571–585. Springer-Verlag, 2009.
5. Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Emmanuel Gaudin, Elie Najm, and Rick Reed, editors, *SDL 2007*, volume 4745 of *Lecture Notes in Computer Science*, pages 166–185. Springer, September 2007.
6. F. A. Kraemer and P. Herrmann. Reactive Semantics for Distributed UML Activities. In J. Hatcliff and E. Zucca, editors, *Formal Techniques for Distributed Systems*, volume 6117 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2010.
7. Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann. Tool Support for the Rapid Composition, Analysis and Implementation of Reactive Services. *Journal of Systems and Software*, 2009.
8. Frank Alexander Kraemer, Rolv Bræk, and Peter Herrmann. Compositional Service Engineering with Arctis. *Telektronikk*, 105(1), 2009.
9. Marius Bjerke. Runtime Support for Executable Components with Sessions. Master's thesis, Norwegian University of Science and Technology, July 2009.
10. Frank Alexander Kraemer, Vidar Slåtten, and Peter Herrmann. Model-Driven Construction of Embedded Applications based on Reusable Building Blocks – An Example. In Attila Bilgic, Reinhard Gotzhein, and Rick Reed, editors, *SDL 2009*, volume 5719 of *Lecture Notes in Computer Science*, pp. 1–18. Springer, 2009.
11. Stephan Haugsrud. Developing Android Applications with Arctis. Master's thesis, Norwegian University of Science and Technology, June 2009.
12. J.M. Almendros-Jimenez and L. Iribarne. Designing GUI Components from UML Use Cases. *Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the*, pp. 210–217, April 2005.
13. Stefan Link, Thomas Schuster, Philip Hoyer, and Sebastian Abeck. Focusing Graphical User Interfaces in Model-Driven Software Development. In *ACHI '08: Proceedings of the First International Conference on Advances in Computer-Human Interaction*, pp. 3–8, Washington, DC, USA, 2008. IEEE Computer Society.

14. Paulo Pinheiro Da Silva and Norman W. Paton. User Interface Modelling with UML. In *In Proceedings of the 10th European-Japanese Conference on Information Modelling and Knowledge Representation*, pp. 203–217. IOS Press, 2000.
15. Peter Braun and Ronny Eckhaus. Experiences on Model-Driven Software Development for Mobile Applications. In *ECBS '08: Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 490–493, Washington, DC, USA, 2008. IEEE Computer Society.
16. N. Koch, H. Baumeister, R. Hennicker and L. Mandel. Extending UML to Model Navigation and Presentation in Web Applications. In *UML*, 2000.
17. S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. *Computer Networks*, 33(1-6), pp. 137–157, 2000.
18. Jaime Gómez, Cristina Cachero, and Oscar Pastor. Conceptual Modeling of Device-Independent Web Applications. *IEEE MultiMedia*, 8(2), pp. 26–39, 2001.
19. P. Caceres, E. Marcos, and B. Vela. A MDA-based Approach for Web Information System Development. *Proceedings of WISME*, 2003.
20. Object Management Group. Service Oriented Architecture Modeling Language (SoaML) - Specification for the UML Profile and Metamodel for Services, November 2008.
21. Nora Koch, Philip Mayer, Reiko Heckel, L. Gönczy, and C. Montangero. UML for Service-Oriented Systems. Technical report, Deliverable 4.14a of the SENSORIA Project, 2007.
22. Benjamin Lieberman. UML Activity Diagrams: Detailing User Interface Navigation. IBM DeveloperWorks Article, 2004. http://www.ibm.com/developerworks/rational/library/4697.html (Accessed December 2010).
23. Jan Van den Bergh and Karin Coninx. Towards Modeling Context-Sensitive Interactive Applications: The Context-Sensitive User Interface Profile (CUP). Proceedings of the 2005 ACM symposium on Software visualization, pp. 87–94, ACM Press, 2005.
24. Google App Inventor Website. http://appinventor.googlelabs.com/ (Accessed December 2010).
25. Wang, Kevin and McCaffrey, Corey and Wendel, Daniel and Klopfer, Eric. 3D game design with programming blocks in StarLogo TNG. In *ICLS '06: Proceedings of the 7th international conference on Learning sciences*, pp. 1008–1009, Bloomington, Indiana, USA, 2006. International Society of the Learning Sciences.
26. Lego Mindstorms Website. http://mindstorms.lego.com/ (Accessed December 2010).