Process-Oriented Behavior Generation using Interaction Patterns

Laura M. Daniele, Luís Ferreira Pires, and Marten van Sinderen
Centre for Telematics and Information Technology
University of Twente
Enschede, the Netherlands
e-mail: {l.m.daniele, l.ferreirapires, m.j.vansinderen}@ewi.utwente.nl

Abstract—**Model-driven approaches have shown that the systematic use of models and models transformations in the design can facilitate the development process of distributed software applications. Abstract models can be used to (automatically) generate other models that gradually add details to an application's structure and behavior, to simulate and execute this behavior in early stages of the development process, to validate it against requirements, or to generate executable code. Since these models document the design at different abstraction levels, they also facilitate the communication between people with different skills, knowledge and background, such as business and IT people. This paper shows how the Business Process Model Notation (BPMN) can be used in a model-driven approach to represent application's behavior at different abstraction levels and reduce the communication gap between different stakeholders in enterprises. The paper focuses on generating executable behaviors, which we represent as BPMN orchestrations, from interaction patterns, which are recurring sequences of actions among interacting components that we can represent as BPMN choreographies.**

Keywords- MDA, behavior modeling, model transformations, BPMN, choreographies, orchestrations, interaction patterns

I. INTRODUCTION

The Model-Driven Architecture (MDA) [18] principles prescribe viewpoints for distributed applications design based on the separation of application functionality from the technology used to implement applications, i.e., the separation of Platform-Independent Model (PIM) and Platform-Specific Model (PSM) concerns. In this way, technology evolution does not affect the PIM design, which can still be reused with other specific PSM technologies. Moreover, the model-driven community promotes the definition of models at different abstraction levels of the design process and the systematic (re)use of transformations between these models. In this way, design knowledge can be used to (automatically) generate more detailed models or executable code from abstract models, to validate application's requirements, or to simulate application's behavior in early stages of the development process. An open issue in the model-driven community concerns how application behavior should be represented [11]. There is agreement on the need to incorporate application behavior at the PIM level of the design process instead of adding this behavior later to the code level [19], and execute this behavior already at the PIM level for evaluation [12,24]. However, there is no agreement on how this should be done, mainly because of the lack of a commonly accepted modeling language to adequately represent behavior [11]. For example, the Universal Modeling Language (UML) [20] is a widespread standard that allows the representation of behaviors as statecharts and activity diagrams. However, UML offers poor support for modeling different levels of abstraction and refinement, and lacks a commonly agreed formal semantics. In contrast, some domain specific languages (DSLs) [9] have formal semantics and are suitable to represent the behavioral aspects mentioned above, but they are still far from a widespread adoption in the community.

The Business Process Modeling Notation (BPMN) [16] is a graphical notation for business process modeling promoted by OMG. Due to its intuitive flow-chart format and syntactic richness, BPMN has emerged as a promising standard notation to bridge the gap between business and IT people in enterprises. Business analysts and managers can use the core elements of BPMN to intuitively model and understand their processes, while IT people can exploit the full expressiveness of BPMN in order to refine these business processes, by adding technical details to generate processes that can be automatically executed by process execution engines. In this way, each stakeholder can address the (same) process at the right level of abstraction, i.e., high abstraction level for business people, and lower abstraction levels for technical developers. Moreover, BPMN 2.0 [17], which is the latest version of the standard, has been enriched with several features. For example, it allows the specification of choreography diagrams to model the abstract behavior of participants in business interactions, and provides a standard mapping to the Business Process Execution Language for Web Services (BPEL) [15], which can be used to execute this behavior.

This paper aims at exploiting the BPMN benefits of being intuitive, widespread and versatile with a broad spectrum of syntactic elements, to exhaustively represent the

behavior of distributed applications at different abstraction levels in the context of MDA. Particularly, the paper shows how BPMN 2.0 can be used as modeling language in a systematic way according to the model-driven approach we defined in [4]. This approach aims ultimately at automatically generating executable orchestrations from abstract choreographies. The paper also outlines some issues that we encountered in this attempt, and points to suitable ways to tackle these issues.

The structure of the paper is the following: Section II presents an overview of our model-driven approach, Section III focuses on a central concept of this approach, i.e., the concept of interaction pattern, Sections IV and V show how BPMN can be used to represent source and target models for our model transformations, respectively, Section VI discusses our choice of using BPMN and outlines the issues we encountered as a consequence of this choice, Section VII discusses some related work. Finally, Section VIII presents our conclusions.

II. MODEL-DRIVEN APPROACH

In our previous work [4], we have defined a model-driven approach based on models at consecutive abstraction levels and model transformations. In this approach, we have first divided our design in two abstraction levels, which are, respectively, a *platform-independent design level* and a *platform-specific design level*. Fig. 1 shows the separation between these PIM and PSM design levels. Since behavioral aspects of applications are sometimes overlooked at the PIM level [11], we focused on the modeling the application's behavior at the platform-independent design level of our approach. In this paper, we relate our levels to the concepts of choreography and orchestration. Fig. 1 shows how our PIM level is decomposed in three models, namely:

- *Service specification (SS)* defines the external observable behavior of the system. At this level, we consider the system as a black box, which receives some inputs from the environment and generates outputs. We do not have yet any knowledge about the internal structure of the system. We consider the SS as a choreography that represents the interactions between the system and its users.
- *Service design refined model (SDRM)* is a refinement of the SS monolithic behavior into a structured behavior. At this level, we consider the system as a set of interacting components, for example, components C_1, C_2 and C_3 in Fig. 1. We consider each of these components as a black box and we do not have yet any knowledge about their internal activities. However, these components interact with each other and we specify these interactions as sequences of message exchanges. For example, the SDRM level in Fig. 1 shows that the input to the system corresponds to an input message I_1 to the component C_1 that generates an intermediate output message $O_1 \equiv I_2$, which is then taken as input by component C_2. The message exchange continues until component C_2 generates the output message O_4, which corresponds to the final output of the system. We consider the SDRM as a choreography that represents the interactions among the components of the system.
- *Service design component model (SDCM)* is a refinement of the SDRM as the detailed executable behavior of concrete components. At this level, we consider the system as a set of interacting components with individual internal processes and activities. Fig. 1 shows that each component has an internal flow of activities in order to provide inputs and outputs for the message exchange. We consider the SDCM as a collaboration of behaviors that fulfills the choreography defined in the SDRM.

As depicted in Fig. 1, our model-driven approach includes two model transformations at the PIM level, i.e., transformations T_1 and T_2, and one transformation from PIM to PSM. Transformation T_1 refines the SS choreography into a more detailed choreography at the SDRM level. Since this choreography is not executable, transformation T_2 refines it into the SDCM orchestration, which can in principle be executed. The transformation from PIM to PSM maps the SDCM, which is platform-independent, onto some specific middleware platform on which the design can be realized. In principle, it is possible to use different middleware platforms to implement the SDCM. A common requirement to all these transformations is that they should preserve correctness and consistency with the original abstract specification of the system. In other words, it is possible to refine these models by gradually adding details to specify the internal view of the system. However, consecutive models should always preserve the original behavior from the perspective of the external environment. This is represented in Fig. 1, by keeping the inputs and outputs arrows external to the system at any level of abstraction: the level of details gradually increases from SS to SDCM, but the inputs and outputs to/from the system should always be the same.

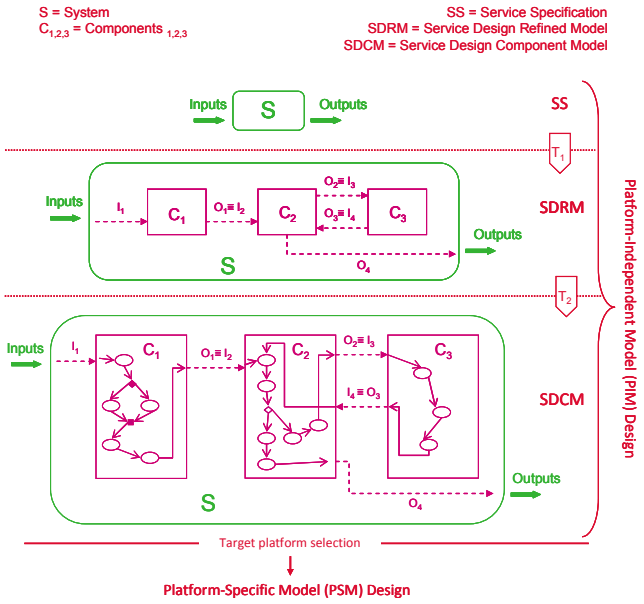

Figure 1. Model-Driven approach and abstraction levels.

This paper focuses on transformations T_1 and T_2 at the PIM level and shows how BPMN can be used as a modeling language to represent the source and target models of these transformations.

III. INTERACTION PATTERNS

Another important design practice consists of collecting the knowledge acquired in some design step and reusing it in other steps of the same design process and/or in the design process of new applications, instead of creating these applications from scratch. In our model-driven approach we follow this practice. Initially, we created our models manually in order to have a clear understanding of the source and target models of our transformations. Afterwards, we created manual mappings from source to target model in order to generate systematic guidelines for these transformations. Finally, we used these guidelines to learn how these transformations could be possibly automated.

During the phase of manual refinement of an SS into an SDRM, we have been able to identify in the SDRM a set of recurrent behavior execution traces among the components of the system. The SDRM can be considered as a choreography that represents the interactions between components. We called the identified traces *interaction patterns*, which we defined as "sequences of actions performed by two or more interacting components defined from the internal perspective of the system" [4]. In this step, we also have been able to identify two different types of interaction patterns, namely *basic* and *composite* patterns. Basic patterns involve interactions between two components, and composite patterns involve interactions between more than two components. Fig. 2 shows an example of basic interaction patterns. On the left side of Fig. 2, the SDRM depicted in Fig. 1 is split in four basic interaction patterns and, on the right side, these patterns are represented using BPMN.

The first situation depicted in Fig. 2 is the interaction between the environment external to the system and the system component C_1. Particularly, the environment provides an input I_1 to C_1. This situation is represented on the right side of Fig. 2 in BPMN as a choreography task, which is the rounded rectangle named *Basic Interaction Pattern #1*, between the two participants *Environment* and *Component 1*, which are the two bands on top and bottom of the choreography task, respectively. The unshaded participant, namely the *Environment*, is the initiator of the interaction and sends a message (*Input* ≡ I_1) to the other participant *Component 1*. Since *Component 1* does not send a message back, this represents a one-way interaction. Analogously, Fig. 2 also depicts: (1) the *Basic Interaction Pattern #2* between the two participants *Component 1* and *Component 2*, (2) the *Basic Interaction Pattern #3* between the two participants *Component 2* and *Component 3*, which is a two-way interaction in which the shaded participant *Component 3* sends back a message ($I_4 \equiv O_3$) to the initiator of the task *Component 1,* and (3) the *Basic Interaction Pattern #4* between the two participants *Component 2* and *Environment*.

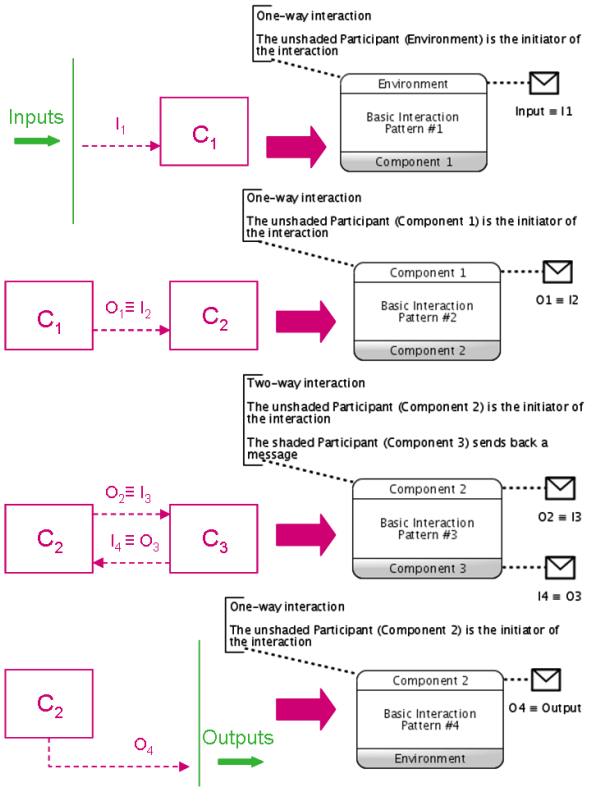

Figure 2. Example: basic interaction patterns.

Fig. 3 shows an example of a composite interaction pattern, which consists of a sequence of basic patterns from Fig. 2. We represented the composite interaction pattern in Fig. 3 as a BPMN choreography sub-process named *Composite Interaction Pattern*. A choreography sub-process is a compound choreography that can be refined into a finer level of detail, i.e., a set of atomic choreographies, which are represented in Fig. 3 in terms of the basic interaction patterns described above. The participants of the choreography sub-process are displayed in the upper and lower bands, and the unshaded participant, namely the *Environment*, is the initiator of the sub-process.

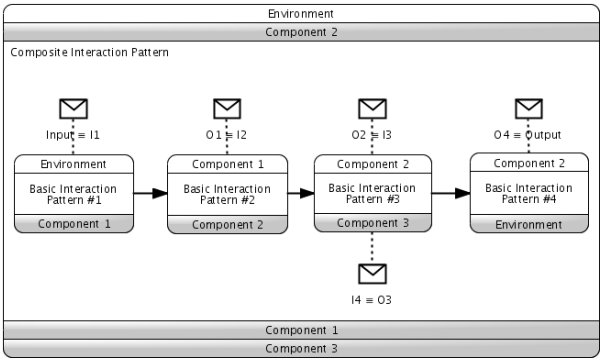

Figure 3. Example: composite interaction pattern.

Although not explicitly represented with a starting and ending event, the composite interaction pattern in Fig. 3 starts with the *Environment* participant of the *Basic Interaction Pattern #1*, which sends the message *Input* ≡ I_1 to the *Component 1*, and ends with the *Environment* participant of the *Basic Interaction Pattern #4*, which receives the message *Output* ≡ O_4 from *Component 2*.

During the phase of manual refinement of the SDRM into the SDCM, we mapped the interaction patterns identified at the SDRM level onto corresponding patterns at the SDCM level. Moreover, we used the design knowledge acquired in the previous step as bottom-up knowledge, i.e., we used the interaction patterns as markers for the SS level. In this way, we created a vertical correspondence of interaction patterns from SS to SDRM to SDCM that can be used to facilitate the automation of the approach. Ideally, we aim at (1) allowing a designer to assemble the behavior of new applications at a high level of abstraction (SS) as combinations of existing building blocks marked as *abstract* interaction patterns, (2) automatically obtaining more refined interaction patterns (SDRM), and (3) generating *executable* interaction patterns (SDCM) that are consistent and correct with respect to the original application behavior.

IV. TRANSFORMATION T_1

This section presents an example of source and target models for transformation T_1 from SS to SDRM. These examples are excerpts of the Live Contacts case study [10,27], which has been defined and applied in the A-MUSE project [1]. Live Contacts is a mobile application that runs on Pocket PC phones, smart phones and desktop PCs, and allows its users to contact the right person, at the right time, at the right place, via the right communication channel. Behavioral aspects of source and target models are represented as BPMN choreography diagrams. These diagrams refer to a UML information model that represents the status information handled by the system. Due to space limitations, we only show the BPMN behavioral models.

A. Source Model (SS)

The SS specifies the interactions between the environment and the system. Fig. 4 depicts some possible interactions between the user, which in this case represents the environment, and the system. Particularly, in Fig. 4 we consider two options: (1) the *Remove Buddy* function, which allows the user to remove one of the buddies from his buddy list, and (2) the *Proximity Event* function, which allows the user to be notified with an alert when a buddy, who is also online in a chat application, is in the neighborhood of the user. Each of these two functions is represented as a choreography sub-process between the *User* and the *System*. The unshaded participant in a choreography sub-process is the initiator of the interaction. Therefore, the *User* is the initiator of the *Remove Buddy* function, and the *System* is the initiator of the *Proximity Event* function.

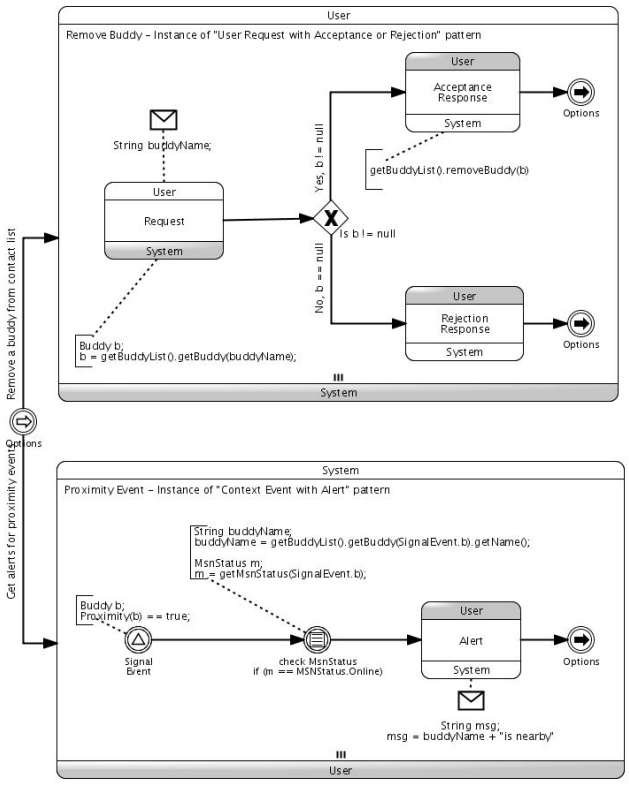

Figure 4. Service specification (SS).

According to our classification of interaction patterns, the *Remove Buddy* choreography sub-process is an instance of the *user request with acceptance or rejection* composite interaction pattern. This pattern consists of a user request to perform a certain task followed by confirmation whether this task has been successfully performed or not. As shown in Fig. 4, the *Remove Buddy* choreography sub-process starts with a *Request* choreography task between the *User* and the *System*. The *User* initiates the interaction by sending a request to the *System* with the name of the buddy to be removed (*String buddyName*). As a consequence, the *System* searches for a buddy object corresponding to the buddy to be removed (*b=getBuddyList.getBuddy(buddy name)* function). This function is specified in our BPMN model as a textual annotation attached to the *System* participant. This is followed by an exclusive gateway (the diamond shape with an X marker) in which the *System* evaluates whether the required buddy is actually in the list of the user (retrieved buddy object is not null). In affirmative case, the gateway is followed by the *Acceptance Response* choreography task, in which the *System* removes the buddy from the list of the user (*getBuddyList().removeBuddy(b)* function). In negative case, namely if the buddy is not the list and cannot be removed, the gateway is followed by the *Rejection Response* choreography task, in which the *System* sends a rejection to the *User*. Both the *Acceptance Response* and *Rejection Response* choreography tasks are followed by an intermediate link throw event named *Options*, which is the circle with a double thin line and a black arrow inside in

Fig. 4. This event loops back to the intermediate link catch event in Fig. 4, also named *Options*, but with a white arrow instead of black. In this way, whenever the *Remove Buddy* reaches either the *Acceptance Response* or *Rejection Response*, the control goes back to the (white arrow) catch link event, from which the options *Remove Buddy* or *Proximity Event* can be chosen again. There can be multiple link throw events (black arrows) in different choreography sub-processes, but only one link catch event (white arrow).

The *Proximity Event* choreography sub-process is an instance of the *context event with alert* composite interaction pattern. This pattern consists of a signal event that catches a context change in the context of the user, followed by a conditional event to check some extra condition, and ending with an alert to the user in case this extra condition is fulfilled. As shown in Fig. 4, the *Proximity Event* choreography sub-process starts with an intermediate signal event (the unfilled circle with a double thin line and a triangle marker inside), named *Signal Event*. Fig. 4 shows in a textual annotation that the *Signal Event* occurs when one of the buddies of the user (*Buddy b*) is nearby the user (*Proximity (b) == true*). This is followed by an intermediate conditional event (*check MsnStatus*), in which the *System* retrieves the *MsnStatus* of this buddy in order to check if it is online. If *m == MsnStatus.Online*, the *System* sends an alert to the *User* to notify that a certain buddy is nearby (*msg = buddyName + "is nearby"*). The *Alert* choreography task between the *System* and the *User* is followed by a (black arrow) link throw event that loops back to the (white arrow) catch link event.

The choreography sub-processes *Remove Buddy* and *Proximity Event* model the interactions between the system and one user of the system. In reality, multiple instances of these choreographies can be executed simultaneously, one for each user of the system. We have indicated this by marking the two choreographies in Fig. 4 as multi-instance sub-processes, i.e., with the marker consisting of a set of three vertical lines and located at the bottom of the rounded rectangle that represents the choreography sub-process.

B. Target Model (SDRM)

In the SDRM, the SS interactions between the user and the system are distributed over the components that constitute the system. These components are specific to the particular application to be developed, i.e., to the specific *reference architecture* that is used to design the system. In this work, we use a reference architecture that is tailored to a family of applications called *context-aware mobile applications*. These are intelligent applications that can monitor the context of their users and, in case of changes in this context, consequently adapt their behavior in order to satisfy the user's current needs or anticipate the user's intentions. We refer to our previous work [4] for a justification and explanation of this reference architecture. In this paper, we only introduce the components that are relevant to understand the examples. Fig. 5 shows an example of SDRM.

Fig. 5 shows that *Remove Buddy* is a choreography sub-process instance of the *user request with acceptance or rejection* composite pattern and consists of five basic interaction patterns, namely *Request*, *Search*, *Update*, *Acceptance Response* and *Rejection Response*. As represented in the upper and lower bands of this choreography sub-process, the components involved in this composite pattern are the *User Agent*, the *Coordinator*, and the *Database*. The user agent component, which acts on behalf of the user with the application, provides user input events to the coordinator component and eventually receives outputs as a consequence of these events. The coordinator component takes care of orchestrating the other components of the applications, searching and updating a database with status information about the users.

The *User Agent* is the initiator of the *Remove Buddy* choreography sub-process. The *Request* basic interaction pattern is a one-way choreography task between the *User Agent* and the *Coordinator*, in which the *User Agent* initiates the interaction by sending a message with the name of the buddy to be removed. This is followed by the *Search* basic interaction pattern, which is a two-way choreography task in which: (1) the *Coordinator* sends to the *Database* a message with the buddy name, and (2) the *Database* retrieves the corresponding buddy object and sends it back to the *Coordinator*. This is followed by the exclusive gateway that evaluates whether this buddy object has been found or not. In affirmative case, the one way *Update* basic interaction pattern occurs, in which the *Coordinator* requests the removal of the buddy to the *Database* (*getBuddyList().removeBuddy(b)* function). This is followed by the one-way *Acceptance Response* basic interaction pattern, in which the *Coordinator* informs the *User Agent* about the successful removal of the buddy. In negative case, the exclusive gateway is followed by the one-way *Rejection Response* basic interaction pattern, in which the *Coordinator* informs the *User Agent* about the unsuccessful removal of the buddy. After either the *Acceptance Response* or the *Rejection Response*, there is a black arrow link event that directs the control back to the white arrow link event (*Options*), from which the options *Remove Buddy* or *Proximity Event* can be chosen again.

Since we assume that the reader is already acquainted with the BPMN elements used so far in this paper, we explain only briefly the *Proximity Event* composite pattern, which is represented in Fig. 5. This pattern consists of a choreography sub-process composed of five basic interaction patterns, namely *Subscribe Event*, *Unsubscribe Event*, *Search*, *Context Query* and *Alert*. As represented in the upper and lower bands of this choreography sub-process, the components involved in this composite pattern are the *Coordinator*, the *Database*, the *User Agent* and the *Context Source*. The context source is the component dedicated to sense changes in the user's context and provides the coordinator component with context events.

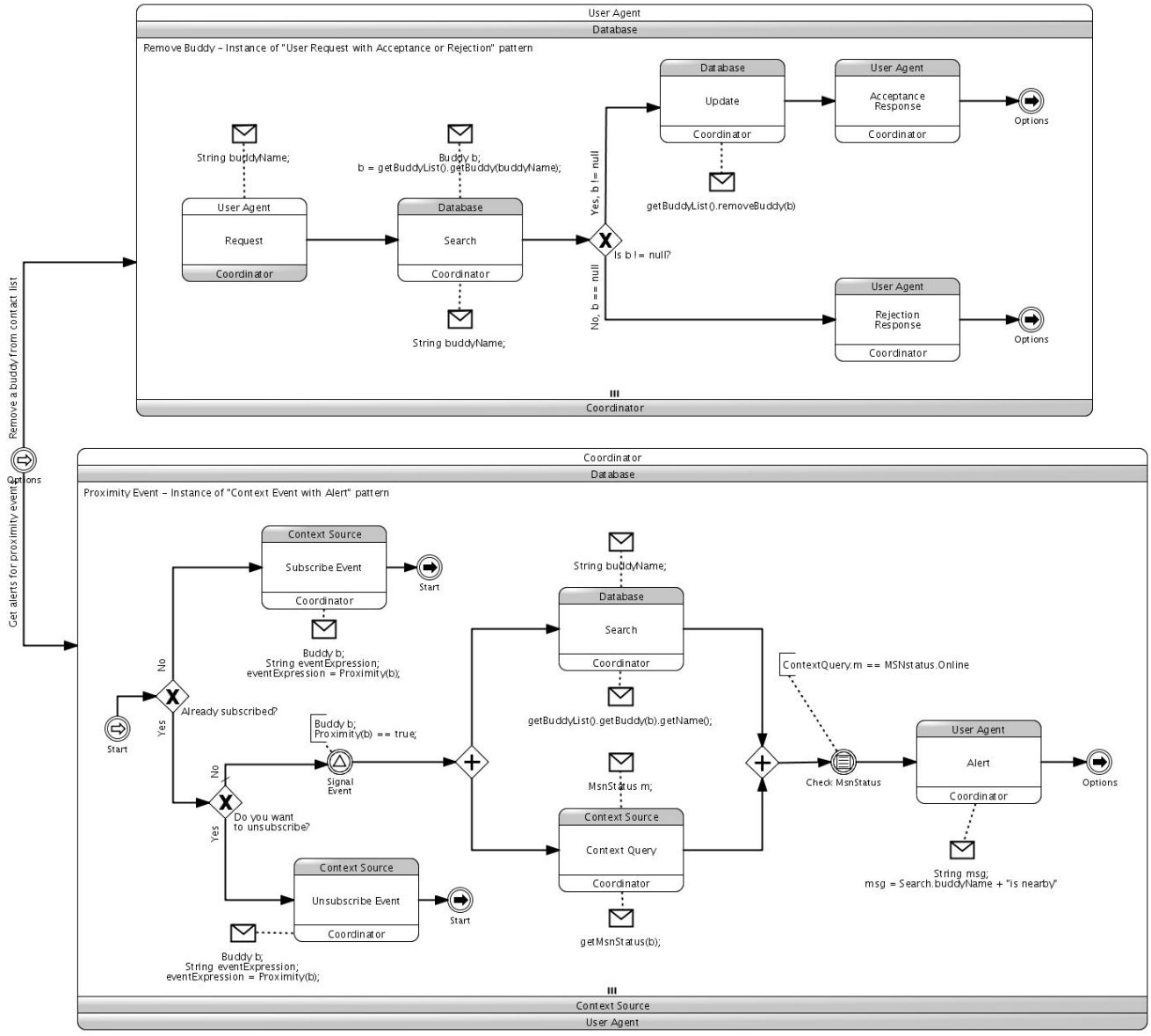

Figure 5. Service design refined model (SDRM).

In order to notify the user about the occurrence of a *Proximity Event*, the *Coordinator* has first to subscribe to a context source for that particular event (*Subscribe Event* basic pattern). Afterwards, the control goes back to the *Start* of the choreography sub-process and the *Coordinator* can unsubscribe for the *Proximity Event* (*Unsubscribe Event* basic pattern) if the user is not interested any more in getting proximity notifications for a certain buddy. Otherwise, the *Coordinator* waits for the occurrence of a *Signal Event* in which the *Context Source* notifies the *Coordinator* when a buddy is nearby the user. When this happens, the *Coordinator* retrieves the buddy name from the *Database* (*Search* basic pattern), and concurrently requests to the *Context Source* the actual chat (MSN) status of that buddy (*Context Query* basic pattern) in order to check if this value is "online". If this is the case (*Check MsnStatus* conditional event), then the *Coordinator* generates an alert to the *User Agent* (*Alert* basic pattern) and the control goes back to the white arrow link event (*Options*), from which the options *Remove Buddy* or *Proximity Event* can be chosen again.

V. TRANSFORMATION T_2

This section discusses transformation T_2 from SDRM to SDCM. The behavioral aspects of source and target models are represented in BPMN. The status information handled in these models refers to an information model represented as UML class diagram that is not shown in this paper due to space limitations. The source model of this transformation is the target model of transformation T_1, i.e., the SDRM of Fig. 5.

Fig. 6 presents an example of SDCM, which is the target model of this transformation. In this SDCM, each component is represented as a detailed (private) process, namely, an orchestration. A composite interaction pattern is represented as a sub-process, which is a compound task that can be refined into a finer level of detail, i.e., a set of atomic tasks that represents our basic interaction patterns. In order to avoid clogging the figure we only represent the *Remove Buddy* instance of the *user request with acceptance or rejection* composite pattern. The *Proximity Event* has a similar representation. Each component collaborates with the other components/processes through the choreography defined in the source model of Fig. 5.

The *Remove Buddy* pattern in Fig. 5 involves the *User Agent*, the *Coordinator*, and the *Database*. Therefore, in Fig. 6 we have three pools that contain the internal activities of the *User Agent*, the *Coordinator*, and the *Database*, respectively. Each of these pools contains a *Remove Buddy* sub-process, which describes the set of internal activities performed within that specific component in order to fulfill the request of removing a buddy from the buddy list of the user.

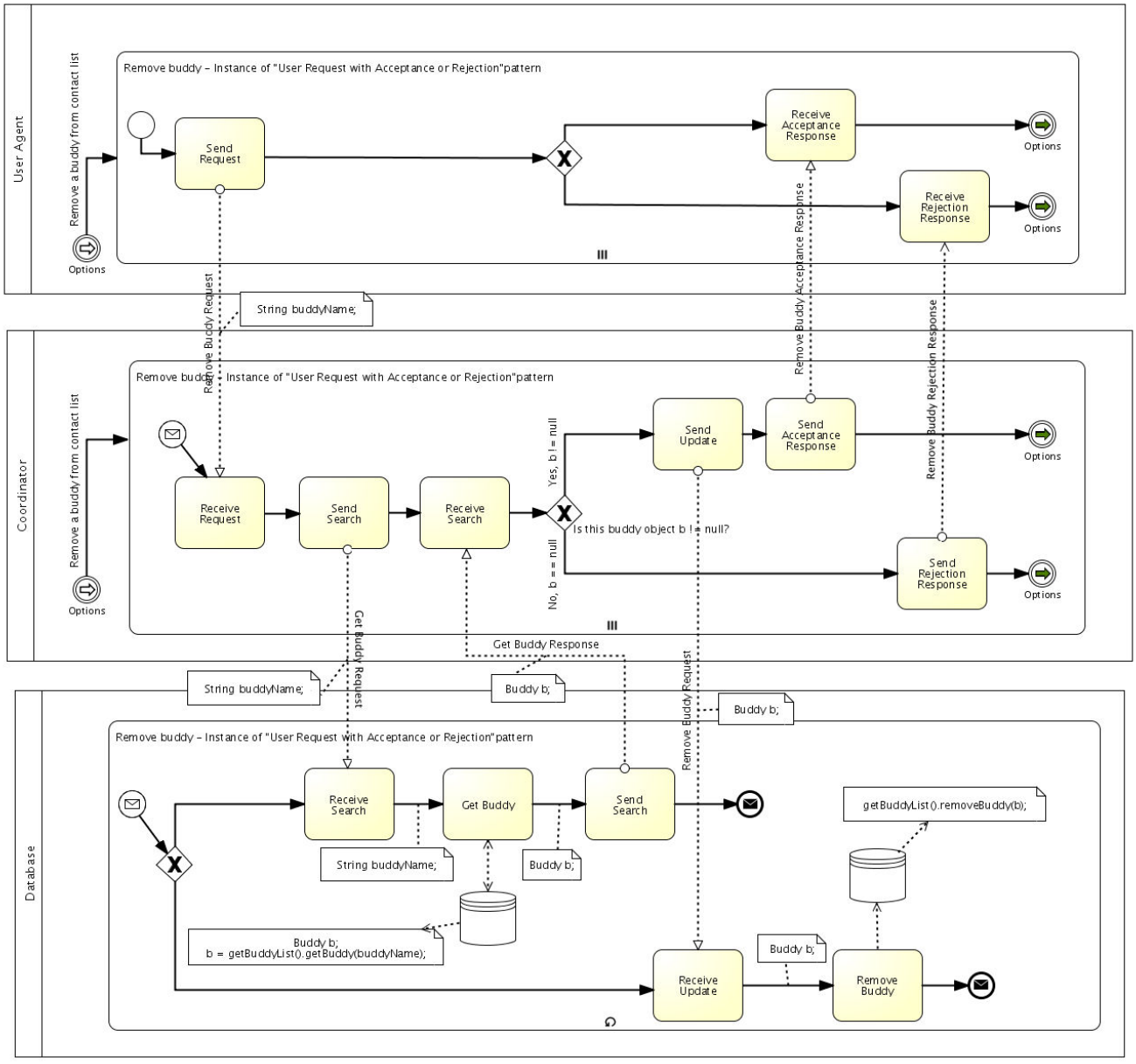

Figure 6. Service design component model (SDCM).

In Fig. 5, the initiator of the *Remove Buddy* choreography is the *User Agent*. This is represented in Fig. 6 with a start event in *the User Agent* process (the white circle within the *Remove Buddy* sub-process). This is followed by a *Send Request* task, in which the *User Agent* sends a message flow (*Remove Buddy Request*) with the name of the buddy to be removed by the *Coordinator*. This initiates the *Remove Buddy* sub-process of the *Coordinator*, which was waiting for a message in order to start. The latter is represented by the start message event, which is the white circle with a message marker inside. Upon the reception of this message (*Receive Request* task), the *Coordinator* sends a request to the *Database* (*Send Search* task) in order to retrieve the buddy object corresponding to the requested buddy (*Get Buddy Request* message flow). After retrieving this buddy from its data store (*Get Buddy* task), the *Database* sends a *Get Buddy Response* back to the *Coordinator* (*Send Search* task). The *Coordinator* receives this response (*Receive Search* task) and evaluates whether the buddy object attached to the response has been found (exclusive gateway with the X marker). In affirmative case, the *Coordinator* sends a *Remove Buddy Request* to the *Database* (*Send Update* task). The *Database* receives the request (*Receive Update* task) and updates the data store (*Remove Buddy* task). This is followed by a *Send Acceptance Response* task, in which the *Coordinator* sends the *Remove Buddy Acceptance Response* message, which is accepted by the *User Agent* (*Remove Acceptance Response* task). In negative case (object not found), the exclusive gateway is followed by a *Send Rejection Response* task, in which the *Coordinator* sends the *Remove Buddy Rejection e Response* message, which is accepted by the *User Agent* (*Remove Rejection Response* task).

In both the *User Agent* and *Coordinator* of Fig. 6, the *Remove Buddy* sub-process ends with a throw link intermediate event named *Options* (the circle with a double thin line with a filled arrow), which redirects the control flow to the catch link event (with unfilled arrow) outside the sub-process. In this way, we create a loop and a new option can be chosen again. Moreover, the *Remove Buddy* sub-process of both the *User Agent* and *Coordinator* in Fig. 6 is marked as a multi-instance process (the three vertical lines marker). We used this marker since there are multiple instances of the *Remove Buddy* sub-process running simultaneously in the coordinator, and each of these instances has a correlation with a particular instance of the user agent (we assume that there is one user agent for each user of the system). We also assume that there is one database for the entire system. Therefore, the *Remove Buddy* sub-process of the *Database* does not have a multi-instance marker. In contrast, it presents a small looping indicator at the bottom of the rectangle that represents the sub-process. In this way, when one of the branches in the *Database* in Fig. 6 reaches the end message event (the circle with a black message marker inside), control flow is sent back to the start message event, which waits for the reception of a new message to start the sub-process again.

VI. DISCUSSION

Since BPMN 2.0 allows the specification of several types of diagrams, such as collaborations, orchestrations, and choreographies, it can be used to represent different levels of behavioral refinement, from abstract specifications to executable designs. Therefore, this allows the same language to be used throughout the whole design process. This is not possible with other modeling languages, such as UML statecharts or activity diagrams, which do not allow us to represent all these abstraction levels. In fact, we could represent the SDRM as a combination of UML statecharts and activity diagrams, while with BPMN we only need one type of diagram, i.e., the choreography diagram. From these UML diagrams, we could generate our SDCM as state machine-based models, as proposed in [28]. The problem is that UML does not give support to define a SS, which is important to model the high level interactions between the system and its users. Alternatively to UML, we can represent our models using some domain specific languages (DSLs), such as the A-MUSE Domain Specific Language to represent the SS and SDRM, and the Interaction System Design Language [9] to represent the SDCM, as we proposed in [4]. However, this would limit our work to a domain specific language that is not commonly adopted. Therefore, we experimented with BPMN in this work. BPMN is already used a lot in both academia and industry, and is a standard supported by different tools from different vendors. We do not claim that BPMN is the only or the best solution to model business processes. We only argue that is a suitable and convenient solution, especially for the purposes of this work.

The use of BPMN as the single notation to represent all our models at different abstraction levels is also beneficial when considering the implementation of the transformations between these models, since all these models conform to the same metamodel. In our approach, we mostly realize architectural transformations that map element structures of a source model to more refined element structures in the target model. For example, the transformation T_1 maps the *Remove Buddy* choreography sub-process of Fig. 4 into the more refined *Remove Buddy* choreography sub-process of Fig. 5. Afterwards, the transformation T_2 maps this refined *Remove Buddy* choreography sub-process onto a more complex collaboration of orchestrations, depicted in Fig. 6. In this way, we have to handle only the BPMN metamodel and, from SS to SDCM, realize our transformations by relating sets of elements within this metamodel.

We encountered some practical problems using BPMN. The current version of the standard is BPMN 2.0, which supports the choreography diagrams we used to represent our SS and SDRM. However, not all currently available BPMN tools support this version yet. For example, at the moment of writing, the BPMN modeler for the Eclipse platform [7] is available only for BPMN version 1.2. Eclipse would be our favorite choice since it is an integrated environment in which we can both edit our models and realize our transformations with Eclipse-based tools, such as ATL toolkit [2] or mediniQVT [13]. For the work presented

in this paper, we used the Signavio/Oryx editor [25], which is a process modeling platform freely available for academic use that supports BPMN 2.0. Concerning our model transformations, the BPMN 2.0 metamodel has been officially made available by OMG in XMI [21]. Some *Ecore* versions of this metamodel available on the Internet allowed us to experiment with the automation of our transformations. Our previous work [4] shows how mediniQVT can be used to automate the transformation from SS to SDRM. Ongoing work addresses the full automation of the transformations described in this paper.

By carrying out this work, we could also identify some open issues that need to be addressed in future. For example, we need to check correctness and consistency of our models, i.e., prove at semantic level that models with different abstraction levels are equivalent. For example, we should prove that the models in Fig. 4, Fig. 5, and Fig. 6, which have gradually increasing levels of detail, are also formally equivalent. In this paper, these models are "correct by construction" since we created them from the definition of the transformations. However, our approach should consider some formalism to check this equivalence for new models that are automatically generated. For example, we foresee that Petri Nets may be a suitable formalism to achieve this goal. In fact, there are already mappings from BPMN to Petri Nets. Therefore, we could check correctness and consistency of our models by using Petri Nets, i.e., by transforming our input and output BPMN models to Petri Nets and then check the equivalence of these Petri Nets models.

Another open issue concerns synchronization and concurrency in the behavior of interacting components at the SDRM and SDCM level. As identified in [5], the main problem with the generation of the SDCM consists of the synchronization and concurrency aspects of interaction patterns that are performed in different threads of control (in parallel), but have functional dependencies. For example, the *Remove Buddy* and *Proximity Event* composite interaction patterns are represented in Fig. 5 as two independent choreography sub-processes, however, the removal of a buddy from the contact list of the user implies that the user is not any more interested in receiving proximity events for that buddy. Therefore, the *Update* basic pattern of the *Remove Buddy* choreography should be followed by the *Unsubscribe* basic pattern of the *Proximity Event* choreography. Further investigation is necessary to determine how these issues can be tackled with BPMN.

VII. Related Work

Some work on behavior modeling in MDA proposes to use state machine-based formalisms as suitable solution to tackle the lack of adequate languages for behavior modeling. In [24], an approach to extend UML with abstract state machines (ASM) is described. This approach, which rises from the need to guarantee executability of behavioral models already at the PIM level, adds the ASM behavior semantic to the UML metamodel, resulting in a combined language called UML+. In [12], state machines are indicated as the most promising basis for capturing and representing system's behavior at the PIM level, and UML notations to represent state machines are discussed. The conclusion in [12] is that these UML notations lacks formal semantics for state machines representation, and a new protocol modeling semantics enriched with some process algebraic constructs is proposed, based on CSP parallel and CCS composition operators. In our previous work [5], we proposed a state machine-based technique that uses Labeled Transition (LTS) and Modal Transition Systems (LTS) to synthesize the behavior of interacting components. We acknowledge that state machines are a suitable formalism for behavior modeling, especially to handle synchronization and concurrency issues, and can be used to validate behavior at the PIM level. However, we find state machines more suitable for the specification of detailed behaviors that are already distributed to components, such as our SDCM, and less suitable for the specification of more abstract behaviors, as in the case of our SS. Moreover, the use of state machines limits the choice in the target platforms that we can use to implement our designs, i.e., it constrains the transformation from PIM to PSM of our approach. A way to derive a PSM from a state machine-based design is the *state pattern* [26], which is a behavioral software design pattern that allows a systematic translation of state machines models to object-oriented programming code, such as Java code. However, if we want to implement our behavioral models using workflow engines, such as BPEL, state machines are not the best choice because the mapping of states and transitions to process-oriented constructs are known to be troublesome. In this case, it is advisable to represent behavioral models in terms of process flow relationships that can be easily processed by these workflow engines. BPMN is a good candidate to achieve this goal, since existing mappings are already available to translate BPMN into BPEL code.

Concerning the mappings of BPMN to other languages, a lot of effort has been spent in the literature not only to map BPMN onto BPEL [14,22], but also to map BPMN onto Petri Nets [6,23]. These mappings are alternatives and at the same time they are complementary to our work. They are alternatives since they realize transformations between different languages, i.e., BPMN to BPEL, or BPMN to Petri Nets. In contrast, we presented mappings that realize (behavioral) transformations within the same language (BPMN). These mappings are complementary since they can be beneficially used in combination with our work. The mappings from BPMN to BPEL provide us with a means to implement our models, i.e., they can be used in the transformation from PIM to PSM of our approach. The mappings from BPMN to Petri Nets provide us with a means to simulate our models and validate their correctness.

We used the concept of interaction patterns to identify recurring sequences of actions performed by two or more interacting components defined from the internal perspective of a service. This is a novel concept that differs from the common use of this term in the literature [8]. A concept rather similar to our concept of interaction pattern is presented in [3], where so called *service interaction patterns* are identified in order to cover multilateral, competing, atomic and causally related interactions. However, the

service interaction patterns in [3] are used in benchmarking in order to identify platform-specific implementation issues, for example, in case BPEL is used. In contrast, our interaction patterns are related to the application behavior at the platform-independent level and are used to refine service specifications that are too abstract to be directly realized by application components. BPEL is a possible platform-specific technology to implement our interaction patterns.

VIII. CONCLUSIONS

In this paper, we have shown how BPMN can be beneficially used in a model-driven approach that models application behavior at different levels of abstraction. We have exploited the rich syntax offered by BPMN to prescribe the use of different type of diagrams in models at consecutive abstraction levels, such as choreography and orchestration diagrams, and we have proposed transformations between these models. These transformations are not fully automated yet. Our ongoing work focuses on the automation of these transformations.

This paper contributes to the attempt of the model-driven community to find adequate languages and formalisms for behavior modeling of applications. These languages should be suitable to be executed so that models expressed in these languages can be validated in early stages of the design process (at the PIM level). The paper also contributes to the attempt of the business process modeling community to bridge the gap between business engineers, who can understand a process at a high level of abstraction, and system developers, who understand the same process at the level of its implementation in terms of technical solutions.

REFERENCES

[1] A-MUSE Project, http://a-muse.freeband.nl.

[2] ATL Project, http://www.eclipse.org/m2m/atl.

[3] A. Barros, M. Dumas, and H.M. ter Hofstede, "Service interaction patterns," Business Process Management, Lecture Notes in Computer Science, vol. 3649, Springer Verlag, 2005, pp. 302-318.

[4] L.M. Daniele, L. Ferreira Pires, and M. van Sinderen, "An MDA-based approach for behaviour modelling of context-aware mobile applications," Proc. of the 5th European Conf. on Model Driven Architecture-Foundations and Applications (ECMDA-FA 2009), Lecture Notes in Computer Science, vol. 5562, Springer Verlag, 2009, pp. 206-220.

[5] L.M. Daniele, L. Ferreira Pires, and M. van Sinderen, "Towards automatic behavior synthesis of a coordinator component for context-aware mobile applications," Proc. of the 13th Enterprise Distributed Object Computing Conference Workshops (EDOCW 2009), IEEE Computer Society Press, 2009, pp 140-147.

[6] R.M. Dijkman, M. Dumas, and C. Ouyang, "Formal semantics and analysis of BPMN process models using Petri Nets," Information and Software Technology, vol. 50(12), 2008, pp. 1281-1294.

[7] Eclipse BPMN modeler, http://www.eclipse.org/bpmn.

[8] Interaction Design Pattern (from Wikipedia), http://en.wikipedia.org/wiki/Interaction_design_pattern.

[9] ISDL home, http://isdl.ctit.utwente.nl.

[10] Live Contacts home, http://livecontacts.telin.nl.

[11] A. McNeile, and N. Simons, "Methods of behavior modeling: A commentary on behavior modeling techniques for MDA," unpublished, http://www.metamaxim.com/download/documents/Methods.pdf.

[12] A. McNeile, and E. Roubtsova, "Composition semantics for executable and evolvable behavioral modeling in MDA," Proc. of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture (BM-MDA 2009), vol. 379, ACM Press, 2009, pp. 1-8.

[13] medini QVT: ikv++ technologies home, http://www.ikv.de.

[14] J. Mendling , K.B. Lassen, and U. Zdun, " Transformation strategies between block-oriented and graph-oriented process modelling languages, " Multikonferenz Wirtschaftsinformatik 2006, vol. 2, GITO-Verlag, 2006, pp.297-312.

[15] OASIS, "Web Services Business Process Execution Language," http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[16] Object Management Group, "Business Process Management Initiative," http://www.bpmn.org.

[17] Object Management Group, "Business Process Model and Notation Specification, Beta 1 for Version 2.0," dtc/2009-08-14, August 2009, http://www.omg.org/spec/BPMN/2.0.

[18] Object Management Group, "MDA guide, V1.0.1," omg/03-06-01, Jun. 2003.

[19] Object Management Group, "OMG Model Driven Architecture: How systems will be built," available at www.omg.org/mda

[20] Object Management Group, "Unified Modeling Language," http://www.uml.org.

[21] Object Management Group, "XML Metadata Interchange (XMI), v2.1.1" http://www.omg.org/technology/documents/formal/xmi.htm.

[22] C. Ouyang, M. Dumas, A.H.M. ter Hofstede, and W.M.P van der Aalst, "Pattern-based Translation of BPMN process models to BPEL web services," International Journal of Web Services Research, vol. 5(1), Idea Group Publishing, 2008, pp. 42–62.

[23] I. Raedts, M. Petkovic, Y.S. Usenko, J.M.E.M. van der Werf, J.F. Groote, and L.J. Somers, "Transformation of BPMN models for behaviour analysis," Proc. of the International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS 2007), INSTICC Press, 2007, pp. 126–137.

[24] E. Riccobene, and P. Scandurra, "Weaving executability into UML class models at PIM level," Proc. of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture (BM-MDA 2009), vol. 379, ACM Press, 2009, pp. 1-9.

[25] Signavio-Oryx Academic Initiative, http://www.signavio.com/en/academic.html

[26] State Pattern (from Wikipedia), http://en.wikipedia.org/wiki/State_pattern.

[27] G.H. Ter Hofte, R.A.A. Otte, H.C.J. Kruse, and M. Snijders, "Context-aware communication with Live Contacts", Conf. Supplement of Computer Supported Cooperative Work (CSCW2004), Chicago, USA, Nov. 2004.

[28] S. Uchitel, G. Brunet, and M. Chechick, "Behavior model synthesis from properties and scenarios," Proc. of the 29th Int'l Conf. on Software Engineering (ICSE 2007), IEEE Computer Society Press, 2007, pp. 34-43.