An MDA-Based Approach for Behaviour Modelling of Context-Aware Mobile Applications*

Laura M. Daniele, Luís Ferreira Pires, and Marten van Sinderen

Centre for Telematics and Information Technology,

University of Twente, Enschede, The Netherlands

{l.m.daniele,l.ferreirapires,m.j.vansinderen}@ewi.utwente.nl

Abstract. Most reported MDA approaches give much attention to structural aspects in PSMs and in generated code, and less attention to the PIM level and the behaviour of the modelled applications. Consequently, application behaviour is generally not (well) defined at the PIM level. This paper presents an MDA-based approach that incorporates behaviour modelling at the PIM level in the development of a specific category of applications, i.e., context-aware mobile applications. The paper also illustrates a behaviour model transformation realized by using *Medini QVT*, which is a tool that implements the Query/ View/ Transformation (QVT) Relations specification defined by OMG for model-to-model transformations.

1 Introduction

The MDA community agrees on the need to consider behavioural aspects in model transformations meant to support application development. However, there is still no agreement on how behaviour aspects should be supported. Although considerable effort has been done on model transformations from PIMs to PSMs in several application domains, most reported MDA approaches [7,8,13] give much attention to structural aspects in PSMs and in generated code, and less attention to the PIM level and the behaviour of the modelled applications. Consequently, application behaviour is generally not (well) defined at the PIM level, and behavioural aspects have to be incorporated later in the development process, by adding hand-written code as annotations to PSMs or to implementation code skeletons. In this paper, we provide an MDA-based approach that focuses on behaviour modelling of a specific category of applications, namely, context-aware mobile applications. Context-aware mobile applications are intelligent applications capable to sense changes in the user's environment and consequently adjust their behaviour in order to provide relevant functionality to their user anywhere and at anytime.

Our MDA-based approach for the development of context-aware mobile applications decomposes the PIM level in three levels of platform-independence, where each consecutive PIM level consists of a refinement of the previous one, stressing the

---

* This work is part of the Freeband A-MUSE Project (<http://a-muse.freeband.nl>). Freeband is sponsored by the Dutch government under contract BSIK 03025.

behavioural aspects. This paper presents our first results towards the automation of this approach, illustrated with a case study that realises a model transformation with the *Medini QVT* tool [17], which implements the Query/View/Transformation (QVT) Relations specification defined by OMG for model-to-model transformations.

The structure of the paper is the following: Section 2 presents an overview of the A-MUSE MDA-based methodology and focuses on the PIM level of this methodology, Section 3 illustrates with a case study the application of Medini QVT in the first transformation step of our approach, Section 4 discusses our second transformation step, Section 5 discusses some related work, and Section 6 presents our conclusions and identifies topics for future work.

2 MDA-Based Approach

According to MDA principles, the A-MUSE design methodology divides the design of distributed applications in different levels of models with different degrees of abstraction and platform-independence. Fig. 1 shows how the levels considered in this methodology can be applied to context-aware mobile applications [1,2].

The *service specification level* describes a context-aware mobile service¹ as a monolithic behaviour from an external perspective. At this level, we specify the functionality that our service offers to its user and we do not consider any structural detail of the service, i.e., we abstract from its internal components.

Fig. 1. A-MUSE design methodology for context-aware mobile applications

¹ The term *service* at this level denotes the observable behaviour of the whole application, as opposed to the use of the term *service* in service-oriented architectures to denote the functionality supported by a service provider reachable from some middleware.

The *platform-independent service design level* describes a context-aware mobile application from an internal perspective revealing a (given) architecture. This architecture, which is described in [4], consists of context sources, action providers, coordination component, service trader and user components. Context sources sense events in the user's environment and provide these events to the coordination component, which consequently triggers actions that are executed by action providers. Context sources and action providers are registered in the service trader in order to be dynamically available to the coordination component. Each user accesses the service through a user component, which provides the user interface and forwards requests to the coordination component. The core of this architecture consists of the coordination component, since it orchestrates all the internal interactions. These interactions are: (i) user requests from user components, (ii) context events from context sources, (iii) actions to be executed by action providers, and (iv) resources registered in the service trader.

The *platform-specific service design level* describes the realisation of a context-aware mobile application in terms of specific target technologies. Several alternative PSMs may implement a PIM as long as correctness and consistency are guaranteed. Therefore, it is in principle possible to use different middleware technologies to realise the platform-specific service design, such as, for example, web services or CORBA, as indicated in Fig. 1.

This paper focuses on the platform-independent design level of the methodology shown in Fig. 1, namely on the service specification and platform-independent service design model and the transformation T_1 between these models. Fig. 1 shows that the platform-independent design phase in the A-MUSE design methodology is decomposed in the service specification and platform-independent service design steps. The platform-independent service design model should be a refinement of the service specification, which implies that correctness and consistency particularly of behavioural issues have to be addressed in the refinement transformation. However, when trying to realise this refinement transformation, we noticed that the gap between service specification and platform-independent service design is rather wide, so that correctness and consistency was hard to guarantee in a single refinement transformation T_1 . Therefore, we introduced an intermediate step in which the service specification behaviour is refined. This intermediate step results in an intermediate design called *service design refined model* and our final PIM is renamed to *service design component model*. The refinement transformation T_1 has been consistently decomposed in two transformations T_1' and T_1'' . Fig. 2 shows the approach we have defined with all the PIM levels and transformations between these levels.

Fig. 2 depicts the starting point of our approach, i.e., the service specification, in which we define the functionality offered by the application to the user in terms of actions and causality relations between these actions. The actions and causality relations at the service specification level are too abstract to be directly realised by platform-specific technologies. Therefore, the second step of our approach consists of defining a service design refined model, in which we refine each of these actions and causality relations into more detailed actions and causality relations that can be directly supported by the realisation platform. In the next step we compare the refinements in the service design refined model in order to identify sequences of actions that recur in

Fig. 2. PIM levels and transformations between these levels

several refinements. We call these sequences of actions *interaction patterns*, which we have defined in [3] as a ‘recurring sequence of actions performed by two or more interacting components defined from the internal perspective of a service’.

Fig. 2 shows two interaction patterns and their assignment to components. The first interaction pattern, which we have marked with the “*” symbol, recurs in the refinement of actions 1 and 3, and is assigned with dashed arrows to the corresponding components (components A and B in Fig. 2). The second interaction pattern, which we have marked with the “☆” symbol, recurs in the refinement of actions 2 and 4, and is assigned with dotted arrows to components C and D. The assignment of interaction patterns to components results in the service design component model, in which we define the behaviour of each individual component. The service design component model provides the input for the platform-specific service design level without imposing any specific implementation choices. For example, the coordinator component can be implemented as a BPEL process, the action providers as web services, and the service trader as a UDDI registry.

In contrast to our approach, most reported MDA development practices [7,8,13] do not consider behaviour refinements at the PIM level and directly start the development process by defining a platform-independent model of the architecture that implements the application (our service design component model). We believe that this is the major novelty of our approach.

3 Transformation from Specification to Design Refined Model

This section illustrates with a case study the transformation T_1' of Fig. 2 (from service specification to service design refined model). This case study is the Live Contacts application [14], which consists of a context-aware mobile application that runs on Pocket PC phones, Smartphones and desktop PCs. Live Contacts allows its users to

contact the right person, at the right time, at the right place, via the right communication channel. More information about Live Contacts can be found in [15].

We implemented transformation T_1' with the Medini QVT tool [17], which consists of a core engine that implements the QVT Relations standard defined by OMG [18], and a graphical debugger and editor to facilitate transformation development. Fig. 3 shows an overview of the Medini QVT transformation approach. Inputs to Medini QVT transformations are: (i) a source and a target metamodel defined in *Ecore*, which is the metamodel type used by the Eclipse Modeling Framework (EMF) [6], and (ii) a source model conforming to the source metamodel. The Medini QVT transformation produces as output a target model that conforms to the given target metamodel.

Fig. 3. Overview of the Medini QVT transformation approach

3.1 Source Model

The service specification consists of two parts: a UML information model that represents status information handled by the modelled context-aware mobile application, and an A-MUSE DSL model that specifies the behavioural aspects of the application. The A-MUSE Domain Specific Language (DSL) [11] is a language developed and applied in the A-MUSE project [9] that allows us to model behavioural aspects in terms of causality relations between interactions without constraining the internal implementation of the application. Our approach is not restricted to any specific modelling language and in this work the A-MUSE DSL is only a vehicle to reach our purposes.

Since we are developing context-aware mobile applications, our service specification actually includes also a UML context model [4], which represents the relevant concepts used by context-aware mobile service components that manipulate context, i.e., context sources and coordinator component of Fig. 1. Transformation T_1' uses this model as the reference for context information.

In our initial PIM we model the interactions between one user instance and the system. This simplification implies that when we realize the service at the platform-specific design level, we have to make sure that the resulting system is able to support different user instances that run simultaneously and consistently. Fig. 4 depicts this service specification perspective.

Fig. 4. Service specification perspective

Fig. 5 presents a service specification example for the Live Contacts application. We refer to the application user as an item *me* of type *FocusUser*. An item is a global behaviour variable that can be accessed in the behaviour that defines the variable. A user may request to remove a buddy from the buddy list (*removeReq*) by giving as input to the application the name of this buddy. If the buddy is not in the list (*IsInList(removeReq.name, BuddyList): Boolean* condition), the user request is rejected (*removeRej*), otherwise (*IsInList(removeReq.name, BuddyList): Boolean* condition) the request is accepted (*removeAcc*) and the buddy is removed from the buddy list of the user (*me.getBuddyList().removeBuddy(me.getBuddyList().getBuddy(removeReq.name))* function).

The user may also contact a buddy using some specific communication means (*contactReq*). In this case, the user must give as input to the application the buddy name and the preferred means to reach this buddy. Depending on the contact means selected by the user, the application opens the appropriate communication channel (SMS or e-mail) and retrieves the mobile number or the e-mail address of the buddy (*getMobilePhoneNr()* or *getEmailAddress()* functions). We do not provide a complete specification of the SMS and Email services, since this is out of the scope of this paper. We only assume that these services need an input (*mobileNr* and *emailAddress*, respectively).

The user may also be notified by the application with an alert (*proximityAlert*) when a buddy, who is online in the application, is in the neighbourhood of the user (*proximityEvent*). The status information depicted in the behaviour of Fig. 5 is defined in the UML information model of the service specification.

3.2 Target Model

The service design refined model consists of a UML information model, and an A-MUSE DSL model that describes the structured behaviour of the modelled application, revealing the interactions among architecture components [4]. Fig. 6 shows these components in the architecture that has been applied in the A-MUSE project to realise

Fig. 5. Service specification: example (exported from the A-MUSE DSL editor)

Fig. 6. A-MUSE architecture for context-aware mobile applications

the Live Contacts application. However, this architecture is general enough and can be reused for other context-aware mobile applications by simply redefining some application-specific components, such as context sources and action providers.

Fig. 6 shows the perspective of a single user with a single buddy. The presentation component takes care of the interactions with the end-user. The user agent (UA, one per user) acts on behalf of the user with the presentation component to obtain user input and present user output, and provides the service coordinator with user input events. The service coordinator (C) takes care of orchestrating the other components, searching and updating a database (DB), which contains information about users (e.g., name, password, preferred contact means and list of buddies). We assume that there is one service coordinator and one database. The service coordinator also interacts with context sources and action providers.

The context sources (CS) sense changes in the user context and provides the service coordinator with context events. Fig. 6 shows the (GPS) location service that provides information about a user's current location, the (MSN) presence service that provides indications whether users registered in the Live Contacts application are available online in the network, and the (Outlook) calendar service that provides calendar information. In this example we assume that there is one (GPS) location service, one (MSN) presence service and one (Outlook) calendar service for each user agent. These services are registered in the service trader.

The action providers (AP) are responsible for performing actions triggered by the service coordinator. Actions represent application reactions to user input events and context events. Fig. 6 shows an SMS service, phone service, e-mail service and chat service, which enable a user to communicate with buddies through, respectively, sending messages, making a phone call, sending e-mails or chatting. We assume that there is one of these services for each user agent. These services are registered in the service trader (ST), which registers all the available context sources and action providers in order to allow the coordinator to discover and invoke them.

Fig. 7 presents the service design refined model that reveals the architecture mentioned above and shows how we have refined the functionality described in the service specification, i.e., *removeReq*, *contactReq* and *proximityEvent*, in terms of sequences of actions.

Each action in Fig. 7 is marked with a label and represents an interaction between two components and the direction of this interaction. In order to avoid clogging the figure, we have not included the status information handled by components. This information is the same as depicted in Fig. 5, but assigned to the proper corresponding refined actions.

Fig. 7 shows that the user request to remove a buddy from his/her list arrives to the user agent (UA), which forwards this request to the coordinator (C). The coordinator checks the database (DB) to determine whether the buddy is included in the buddy list of the user (*findRemReq* and *findRemRsp*). If this is the case, the coordinator removes the buddy from the list (*removeBuddy*) and sends a positive response to the user agent (*removeAcc*), which presents the result to the user. If the buddy is not in the list, the coordinator sends a negative response to the user agent (*removeRej*), which presents the result to the user.

The user request to contact a buddy with a specific means arrives to the user agent (*contactReq*). The user agent forwards this request to the coordinator, which retrieves the buddy to be contacted from the database (*findContactReq* and *findContactRsp*).

Fig. 7. Design refined model: example (exported from the A-MUSE DSL editor)

The coordinator evaluates the parameters of the contact request. Depending on the means selected by the user, the right communication channel is selected (SMS or e-mail). In both cases, the coordinator performs two activities concurrently, namely, retrieving from the database the number or address where to contact the buddy, and asking the service trader (ST) to discover the proper service to contact the buddy. Once both activities are concluded, the coordinator is able to invoke the proper action provider (AP) and provide it with the necessary input, which may be the mobile number or the email address of the buddy.

In order to notify the user of the occurrence of a proximity event, the coordinator has to subscribe in the context sources (CS) a context expression related to that event (*subscribeProximity*). When the value of this expression becomes true, the context source notifies the coordinator (*proximityChange*). While retrieving the buddy name from the database (*findBuddyReq* and *findBuddyRsp*), the coordinator requests to the context sources the current MSN status of the buddy (*msnStatusReq* and *msnStatusRsp*) in order to check if this value is *online*. If this is the case, the coordinator generates an alert to the user agent, which forwards the alert to the user (*proximityAlert*).

3.3 Transformation Relations

Most reported MDA model transformations are realised between models that conform to structurally similar but different metamodels, such as, for example, from UML class diagrams to Java code skeletons. In contrast, transformation T_1' is an architectural

transformation, which relates element structures of the source model to more complex element structures in the target model. In order to generate the target model of Fig. 7 from the source model of Fig. 5, we have created a transformation called *SStoSDRM* between the domains *SS* (Service Specification) and *SDRM* (Service Design Refined Model). This transformation consists of five basic relations in the QVT Relations language used by the Medini QVT tool. Fig. 8 shows these relations schematically.

The first relation (*BehaviourMapping* in Fig. 8) creates a one to one mapping of an *SS* element *Behaviour* with name *ServiceSpecification* onto an *SDRM* element *Behaviour* with name *ServiceDesignRefinedModel*. The second relation (*EntryItemInstanceMapping* in Fig. 8) creates a one to one mapping from *SS* to *SDRM* of: (i) an *EnablingRelation* element that relates an *EntryPoint* to an *OrSplit* ($\triangleright \rightarrow \diamond$), (ii) an *EnablingRelation* element that relates an *OrJoin* to the *EntryPoint* of a behaviour instance ($\square \rightarrow \triangleright$), and (iii) an *item* element with name *me*. The third relation (*RemoveRequestMapping* in Fig. 8) maps the *removeReq* functionality of the *SS* onto the

Fig. 8. Transformation relations

refinement of the same functionality in the SDRM. The fourth relation (*ContactRequestMapping* in Fig. 8) maps the *contactReq* functionality of the SS onto the refinement of the same functionality in the SDRM. The last relation (*ProximityMapping* in Fig. 8) maps the *proximity* functionality of the SS onto the refinement of the same functionality in the SDRM. In this way the architectural refinement alternatives are documented and can be applied appropriately to the source model.

4 Transformation from Design Refined Model to Component Model

This section discusses the source and target models of transformation T_1'' of Fig. 2 (from service design refined model to service design component model) with the Live Contacts application.

4.1 Source Model

The source model of this transformation consists of the target model of transformation T_1' . In this model, we have been able to identify recurring sequences of interactions between components (interaction patterns). Particularly, we have identified two types of interaction patterns, namely *basic* and *composite* patterns. Basic interaction patterns occur between two interacting components. An example of basic patterns in Fig. 7 is *findRemReq* and *findRemRsp*, which we call *search* pattern, and involves interactions between the coordinator (C) and the database (DB). Composite interaction patterns occur between more than two components and consist of combinations of basic patterns. Examples of composite patterns in Fig. 7 are the refinement of the *removeReq* functionality, which we call *user request with acceptance or rejection* pattern, the refinement of the *contactBuddy* functionality, which we call *user request with external service pattern*, and the refinement of the *proximityEvent* functionality, which we call *context event with alert* pattern. From our experience with the Live Contacts application we have defined a library of basic and composite patterns [5] that cover the functionality of context-aware mobile applications that comply with the architecture of Fig. 6.

4.2 Target Model

Fig. 9 shows an example of a service design component model with limited functionality. This should be the target model of the transformation T_1'' . In this model, we have assigned the basic and composite interaction patterns identified in the service design refined model to concrete components that can realise these patterns.

Fig. 9 depicts the assignment of the *user request with acceptance or rejection* composite pattern to components. Dashed lines indicate the assignment of basic patterns to components. The three involved components are the user agent, the coordinator and the database. Fig. 9 depicts the behaviour of these components considering their interactions. Fig. 9 uses ISDL (Interaction System Design Language) [12], which allows the specification of behavioural aspects of interacting components.

Fig. 9. Service design component model: example (exported from Grizzle [10])

5 Related Work

We briefly discuss here some related work that deals with MDA approaches in several application domains. In [16] the importance of behaviour modelling techniques for MDA is stressed. Given that MDA requires application behaviour to be explicitly represented at the PIM level and that there is no consensus on how to represent this behaviour, different categories of behaviour modelling methods are analysed. The state-machines paradigm is identified as the best basis for such representation. We agree that state machines are suitable to represent behavioural aspects of applications, but we think that they are more suitable for the specification of behaviours that are directly assigned to concrete components, and less suitable for the specification of highly abstract behaviours, as in the case of our service specification (initial PIM).

Considerable effort has been spent on model transformations from PIMs to PSMs in several application domains. For example, in [13], a formal MDA approach for the development of mobile health system is discussed. A model-driven approach for the development of access control policies for distributed systems is presented in [8]. In [13] an MDA approach to implement personal Information Retrieval (IR) processes is proposed. Although these approaches are applied to different application domains, they all report on case studies in which little attention is given to the behaviour of the modelled application at the PIM level, and directly generate PSMs based on the

architecture that implements the application. The PIMs are defined by using UML, and PSMs are represented in programming languages, such as Java. Therefore, the PIM-to-PSM transformation is performed between models that have structurally similar but different metamodels. In contrast, we focus on application's behaviour refinements at the PIM level and we realise transformations between models that have the same metamodel, but our transformations embody architectural decisions and preserve correctness and consistency.

6 Conclusions and Future Work

According to MDA principles, the A-MUSE design methodology divides the design of distributed applications in platform-independent and platform-specific design levels. In this paper, we focused on the platform-independent design level of this methodology, and we proposed an approach to model the application behaviour at this level through model refinements. While realising these refinements, we noticed that the gap between abstract specifications and concrete models of application components is rather wide, so that it was advisable to add an intermediate level of behaviour modelling to the A-MUSE methodology. Therefore, we defined an approach that decomposes the platform-independent design level in three models and two model transformations. This paper focuses on the first of these model transformations.

Towards the automation of the first model transformation, we investigated the usage of the Medini QVT tool, which allowed us to define transformation rules in the QVT Relation language defined by OMG. Although the results we have discussed in this paper are preliminary, we can already conclude that it is possible to realise automated behaviour model refinements in the context of MDA, and Medini QVT is an appropriate tool to support this goal. However, further work needs to be performed in order to achieve full automation of this transformation. The presented example transformation considers only part of the functionality offered by the Live Contacts application to its users, and it should be extended with transformation rules that consider the whole functionality. These extended transformation rules should also be tested and validated with new case studies in order to demonstrate that they can be reused with several context-aware mobile applications. Moreover, we still have to identify the most appropriate level of granularity to define our transformation rules. When realising transformation T_1' , we have identified two 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. We defined transformation rules for composite interaction patterns and we noticed that these patterns are not very flexible, since they are complex and have a fixed structure. Therefore, our transformation rules became large and complex. The library of composite patterns is small, so the benefit of using these patterns is that the number of transformation rules we needed to create was also small. In contrast, the library of basic patterns is large and it would require a large number of transformation rules. However, basic patterns give more flexibility in the design, since they are small, simple, and can be dynamically combined in different configurations of complex behaviours. Therefore, we learned that when considering the granularity of interaction patterns, the trade-off between the number, size, and complexity of transformation

rules on one hand, and the flexibility of design choices on the other hand have to be considered. The most appropriate level of granularity level for interaction patterns is subject to further investigation.

The second model transformation has been realised manually by assigning interaction patterns to components that realise these patterns. This assignment was quite straightforward, since we defined the interaction patterns as annotated actions that explicitly specify which components participate in the pattern. However, we noticed that some synchronization and concurrency issues of interacting components still had to be considered. For example, the coordinator component that orchestrates all the interactions with other components in Fig. 7 has to schedule somehow the execution of the composite patterns that refine the *removeReq*, the *contactReq*, and the *proximity* abstract actions. The designer may decide to interleave these composite patterns, by executing all the patterns one at a time in a single thread of control. Alternatively, the designer may decide to execute these patterns in parallel threads of control. Independently of the option chosen, some formalism should be used to represent and analyse these choices. Moreover, the model in Fig. 7 represents only one user instance interacting with the application. In reality, the coordinator has to handle multiple user instances running at the same time. Therefore, further investigation is necessary on the formalisms that can be used to support these aspects. Process algebra and labelled (modal) transition systems seem to be suitable formalisms for these purposes. For example, an automated technique for synthesizing behavioural models from safety properties and scenario-based specifications by using Modal Transition Systems (MTSs) that can be directly derived from labelled transition systems is discussed in [19].

Although in this paper we made some design decisions to illustrate the case study, our approach is not restricted to these decisions. For example, we used A-MUSE DSL and ISDL to define our behaviour models, since these are general-purpose languages that allow the modelling of application behavioural aspects in terms of causality relations between interactions without constraining the internal implementation of the modelled application. However, we consider A-MUSE DSL and ISDL as vehicles to define behaviour models that can be automatically transformed. Other languages and techniques, such as UML activity diagrams or Message Sequence Charts (MSC), may be used to define our models, as long as they allow us to represent behavioural aspects of the modelled applications exhaustively. Moreover, we tailored our approach to the design of a specific category of applications, i.e., context-aware mobile applications. However, the same approach for behaviour modelling of applications through model refinements, transformation rules and interaction patterns can be applied to other categories of distributed applications by simply adjusting the reference architecture of Fig. 6. This adjusted reference architecture should reflect the architectural components and interactions between components that hold in the new target application domain. Consistently, the interaction patterns should be identified in accordance with this new architecture.

References

1. Almeida, J.P.A., Iacob, M.E., Jonkers, H., Quartel, D.: Model-Driven Development of Context-Aware Services. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, pp. 213–227. Springer, Heidelberg (2006)

2. Almeida, J.P.A.: Model-Driven Design of Distributed Applications. Ph.D. thesis, University of Twente, Enschede, The Netherlands (2006)
3. Daniele, L., Ferreira Pires, L., van Sinderen, M.: Interaction Patterns for Refining Behaviour Specifications of Context-Aware Mobile Services. In: Proceedings of the 4th International Workshop on Model-Driven Enterprise Information Systems (MDEIS 2008), Barcelona, Spain, June 2008, pp. 64–76. INSTICC Press (2008)
4. Daniele, L., Ferreira Pires, L., van Sinderen, M.: Context Handling in a SOA Infrastructure for Context-Aware Applications. In: Proceedings of the 2nd International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing (ACT4SOC 2008), Porto, Portugal, July 2008, pp. 27–37. INSTICC Press (2008)
5. Daniele, L., Ferreira Pires, L., van Sinderen, M.: Live Contacts Case: from Service Specification to Service Design. A-MUSE project deliverable D2.27 (2008)
6. Eclipse Modeling Framework Project (EMF),
<http://www.eclipse.org/modeling/emf>
7. Eissen, S.M., Stein, B.: An MDA Approach to Implement Personal IR Tools. In: 18th International Conference on Database and Expert Systems Applications (DEXA 2007), pp. 259–263. IEEE Computer Society Press, Los Alamitos (2007)
8. Fink, T., Koch, M., Pauls, K.: An MDA Approach to Access Control Specifications Using MOF and UML Profiles. In: The First International Workshop on Views on Designing Complex Architectures (VODCA 2004). Electronic Notes in Theoretical Computer Science, vol. 142, pp. 161–179 (2006)
9. Freeband A-MUSE Project, <http://a-muse.freeband.nl>
10. Grizzle Home, <http://isdl.ctit.utwente.nl/tools/grizzle>
11. Heerink, L., Quartel, D.: Domain Specific Language for Context-Aware Mobile Services. A-MUSE project deliverable D1.13 (2007)
12. ISDL Home, <http://isdl.ctit.utwente.nl>
13. Jones, V., Rensink, A., Ruys, T., Brinksma, E., van Halteren, A.: A Formal MDA Approach for Mobile Health Systems. In: Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA 2004), Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK, Canterbury, pp. 28–35 (2004)
14. Live Contacts Home, <http://livecontacts.telin.nl>
15. Ter Hofte, G.H., Otte, R.A.A., Kruse, H.C.J., Snijders, M.: Context-Aware Communication with Live Contacts. In: Conference Supplement of Computer Supported Cooperative Work (CSCW 2004), Chicago, USA (November 2004)
16. McNeile, A., Simons, N.: Methods of Behaviour Modelling: A Commentary on Behaviour Modelling Techniques for MDA. Metamaxim Ltd Home,
<http://www.metamaxim.com/download/documents/Methods.pdf>
17. Medini QVT: IKV++ Technologies Home, <http://www.ikv.de>
18. Object Management Group: MDA-Guide, Version 1.0.1, omg/03-06-01 (2003)
19. Uchitel, S., Brunet, G., Chechick, M.: Behaviour Model Synthesis from Properties and Scenarios. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, USA, May 2007, pp. 34–43. IEEE Computer Society Press, Los Alamitos (2007)