Behaviour Modelling and Transformations for Context-Aware Mobile Applications

Laura Maria Daniele

Centre for Telematics and Information Technology

Enschede, The Netherlands, 2011

CTIT Ph.D. Thesis Series, No. 11-193

Cover Design: Giovane Moura
Cover photo: façade of Santa Maria Novella (Florence), designed by Leon Battista Alberti
Book Design: Lidwien van de Wijngaert and Henri ter Hofte
Printing: Ipskamp, Enschede, The Netherlands

CTIT Ph.D. Thesis Series, No. 11-193
Centre for Telematics and Information Technology, University of Twente
P.O.Box 217, 7500 AE Enschede, The Netherlands

ISSN 1381-3617; No. 001
ISBN 978-90-365-3204-4

BEHAVIOUR MODELLING AND TRANSFORMATIONS FOR CONTEXT-AWARE MOBILE APPLICATIONS

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof.dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Wednesday, 1st of June 2011 at 14.45

by
Laura Maria Daniele
born on 20th of July 1980
in Cagliari, Italy

Abstract

Today's panorama of service offerings is characterised by the widespread diffusion of the Internet and Web-based technologies everywhere in society. We are surrounded by devices that can support us in several tasks of our every-day life, like, for example, e-readers to access books and magazines, or mobile phones with extremely intuitive user interfaces for browsing, checking emails, keeping in touch with colleagues and friends through social networks, finding maps and locations, and so on. Moreover, this daily experience would not be possible without ultra-fast networks and wireless technologies that allow us to exchange any kind of data, anywhere, in real time and at low-cost. In this panorama, it becomes of vital importance for service providers to offer services that are innovative and distinctive. On one hand, service providers have to preserve current customers and attract new ones in order to survive in an ever growing arena of competitors. On the other hand, service users become more and more aware of the opportunities offered by the evolving technologies and, consequently, more demanding and with stronger expectations than in the past. Therefore, users expect a profusion of services wherever they are, to support whatever they are doing, and according to their personal preferences and needs, while providers have to create a wide range of enriched services in a rapid, low-cost and user-centric way.

This thesis proposes a *layered methodology* based on *behaviour modelling and transformations* for the development of *context-aware mobile applications*, which are distributed applications that can provide advanced and personalised services to their users. Currently available approaches, such as Service-Oriented Architecture (SOA) and Model-Driven Architecture (MDA), are used in this thesis to support such a methodology. The main objective is to progress the state-of-the-art in model-driven development of context-aware mobile applications by taking into account the behaviour of these applications already in early stages of the development process. In order to achieve this, we refine the application behaviour in several steps, from abstract specifications to final implementations, and develop *automated model transformations* throughout these refinement steps to generate executable models and reason about their behavioural correctness.

Acknowledgements

Like the models I talk about in this thesis always have a purpose, I strongly believe that every person I meet in my path has a purpose in the story of my life. It can be really difficult to figure out what this purpose is and, sometimes, I do not know if it is even necessary to do so. What I know is that in these four years of PhD I met a lot of people, whose purpose in my life I already (more or less) figure out, and I would like to express my gratitude to all of them.

First of all, I would like to thank the members of my PhD committee, starting from the ones furthest away: dr. Patricia Dockhorn Costa, dr. Richard Soley, prof.dr. Gregor Engels, prof.dr Paul Grefen, prof.dr. Chris Vissers and prof.dr. Jos van Hillegersberg. It is an honour for me to have each of you in my defense committee. Thank you for accepting the invitation and providing me with your precious feedback to sharpen my thesis. I really appreciate the time and effort you dedicated to my work.

I would like to thank my promotor Mehmet Aksit for his support in these years and the freedom he gave me to carry on with my research trajectory after moving to the Software Engineering group in the first year of my PhD. I also would like to thank my daily supervisors, Luis Ferreira Pires and Marten van Sinderen. I met Luis when I first came to Netherlands in 2005 for my Master thesis as an Erasmus exchange student, I could barely speak English at that time! From that moment, a productive collaboration started, which became even more productive later, in February 2007, when I got my PhD position at the ASNA goup and Marten became my daily supervisor together with Luis. I was lucky to keep their supervison after the ASNA dismissal and I am proud that the three of us, despite the difficulties, managed to collaborate until today and achieve the results presented in this thesis. Luis, thank you for believing in me since the beginning, encouraging me to start the PhD experience, dedicating to me your constant and patient guidance, and flooding my papers and thesis with the ink of your famous and dreaded red pen! Marten, thank you for being always positive, for your persevering encouragement to write about my work and present it all around the world, for promoting a great atmosphere in our working environment and organising memorable events in your house. In this respect, my thought goes to your wife Klaire, who is simply fantastic with me! Klaire, I really enjoyed your company in several occasions, such as the trip to Valencia, and I always felt very welcome in your house. Thank you for that.

Talking about fanstastic people, I have to mention my paranymphs Eduardo and Luiz Olavo, who have walked next to me along this path since the beginning, witnessing every (good and bad) moment of my PhD life. Eduardo, sharing the office with you was a great pleasure. In my mind, I picture you working hard, listening to music with your big headphones and, sometimes, also singing without realising that! I will certainly remember the long and constructive conversations we had about your research or mine, in which just by explaining the problem to the other the solution was already much clearer! Thanks for being this great office mate and friend, always available to listen and help whenever I needed, about work and personal life. Luiz Olavo, spending these years with you has been so much fun! You are such a sociable and enjoyable person, knowledgeable source of gossip, excellent company for travelling, challenging competitor for eating…and you are also an impressive researcher! Definitely, my PhD life wouldn't have been the same without you as a friend and colleague. You are very important to me and I know I can always count on you, thank you!

I started my PhD at the ASNA group, where I met a lot of nice people and incredible researchers. I would like to thank all of them for the amazing environment and the example of dedication and perseverance they offered to me. Among them, I want to thank Patricia, who played an important role in my achievements as a researcher. Patricia, I learned so much from our collaboration during my Master thesis, and I always considered you as an example to follow. When I did not know it yet, you helped me to understand that research was the right direction for me. You motivated me in the choice of the PhD and after four years, I can tell you, the PhD was the best thing could happen: there is no single moment in which I regret my choice! I am also very grateful to Ricardo, Rodrigo and Tom. Ricardo, you are a good friend and colleague since the Erasmus experience and we did a lot of nice things together. It is a pity that eventually you and Kasia had to move away, but I am happy to keep the friendship of both of you despite the distance. Rodrigo, we started the PhD together and it was so easy to get along with you. You immediately became a very good friend of mine. I enjoyed a lot our endless conversations about work and life, sitting outside the Zilverling! Also, many thanks to you and Lorena for the amazing time I had in Brazil: you made me feel at home, now I look forward to having you both in Europe soon. And Tom, my Dutch friend (although very much influenced by our “Latin” culture), many thanks for your support and friendship in these years!

The ASNA dismissal was a big disappointment and, for some time, we did not know what would happen with us. Later, Luis, Eduardo, Luiz Olavo and I moved to the SE group and, fortunately, we managed to find our place there. I would like to thank the colleagues of the SE group for helping us in the integration process. Especially, I would like to thank Ismenia, Ivan, and Arda. Ismenia, I enjoyed very much talking to you, sharing opinions and exchanging experiences. Ivan, our part of the corridor was a lot of fun because of you! Thanks for the many conversations about research, traditions and politics in our countries (how many jokes about Berlusconi I had to take in these years!), and cats (Ciccio Maria thanks you for becoming the most famous cat from Sardinia!). Arda, in this last year, when staying at the university until late for writing my thesis, I had the opportunity to know you better. Thank you for our chats in the dark evenings and, above all, thank you for cheering me up in the worst and most desperate moment of my writing. I also would like to express my gratitude to our secretary Jeanette, extremely helpful concerning bureaucratic issues, and always sensitive to our personal problems. Thanks also to my students Chris Weeink and Steven Bosems for their collaboration. Their final projects contributed to the work in this thesis.

I also would like to thank the colleagues and friends of the DACS group: Anna, Anne, Anya, Desi, Giovane, Idilio, Marijn, Martijn, Rafael, Ramin, Rick, Stephan, Tiago and all the others, who kindly adopted me for the coffee break! I want to give thanks especially to Anna: with the time you became a reference point for my life in Enschede. We are a great example of how people with different personality and origin (you, from the Dolomiti mountains of Belluno, and I, from the Mediterranean sea of Sardinia) can get along and become good friends (in the flat Netherlands!). And, of course, how can I forget Desi! Thank you for your company Desi, I enjoyed our cultural activities and the nice conversations with you. Then, I would like to express my gratitude also to Aiko. Thanks for the amazing trip to Venice: you gave me the opportunity to visit this city like I never did in the past (the fanstastic hotel in Rialto helped a lot!). And thanks for teasing me during these years! I trained myself to weigh my reactions and turn provocative statements into constructive discussions.

In these years, I met a lot of other nice people, like Damiano, Emmanuele, Sandro, Lilya, Elfi, and Maral. Thank you all for the nice time! I also would like to thank Roberto (Coppolecchia) for his amazing help in these years I spent in the Zilverling building at UT. Many thanks also to the Italian community: Andrea, Vanessa, Diego, Jacopo (especially to you, thanks for the support in these last months of writing), Maurizio, Raffaella, Valentina and many others. And to my special (Erasmus) friends Ester,

Rafa, Steffy, Rebe, Sopa and Mariano: no matter where life brings us, we will always be connected!

The extended Brazilian community always played an important role in my life in Enschede: Patricia and JP, Flavia and Pablo, Ricardo and Kasia, Eduardo and Nayeli, Rodrigo, Lisandro, Tiago and Liga, Rafael and Sanijka, Idilio and Suen, Eduardo Z., Ricardo S., Ramon and Rita, and many others. Thank you for the barbeques, parties, drinking nights, and all the memorable events we had together. Above all, I would like to thank Giovane, Luiz Olavo, Luciana and Anna Martha: you are very special to me and I consider you my family here, grazie di tutto, vi voglio tanto bene! Giovane, thanks a lot also for the great job you did with the cover of this thesis, I really appreciated that!

In all these years of PhD I lived in Macandra where I had the opportunity to meet amazing people: I would like to thank all of them for the great time. Most of these people already left, but Juan Carlos, Mehmet, TJ, and Paula are still there: thank you guys, it is always nice to spend time with you. My Italian gang has a special role, namely “compare” Salvo, my personal trainer Ruben, and the barman Stefano: you really made the difference in my life in Macandra. I also want to thank Dana, Emilia, Kasia (Czapinska) and Kasia (Worek) because they are very important friends to me: you really know how to give value to friendship. And of course Uros and Dimitrios. Dimitri, you made me laugh, so much fun and good memories with you. What we had together was special. But you also made me cry. And when you made me cry, it was a lot. Despite all the tears, I wouln’t be the person I am today without you, thank you for everything. You are still very important to me and I am happy we can share this moment: you are part of this! I also would like to thank your family, Lynette and Elias for their hospitality in Greece, and Kathy and Willy for their hospitality in UK, I had fantastic time, thank you all.

When you leave your home town and go to live abroad a lot of things change, but not your best friends. Vale, since the trip in USA when our friendship started, you became essential in my life. I don’t know how you do it, but as soon as I start talking you understand what I feel and give me the right piece of advice. Thanks for the irreplaceable support you gave me in these years despite the distance! Silvy, we started as inseparable volleyball mates long time ago. Now we are not volleyball mates anymore (what a pity, I miss playing volleyball so much!), but we are still inseparable, sharing personal and professional achievements, as well as difficulties and disappointments of life. I am proud of the persons we became and of our friendship, thank you for being always there! Cri, you are my best friend since…ever. I have difficulties to summarise what we have done together, it is so much! Whatever we did in life, we always shared it. In these years I spent in the Netherlands, we had daily chats on MSN, sometimes just to say hello and continue working, and often we had long conversations on Skype. You were far away, but I felt you with me in every moment. I will never thank you enough for who you are and what you do for me. I also would like to thank my friends Raffa and Enrica, it was always great to see you when I was back to Cagliari.

I want to dedicate some words also to Paolo, Barbara, Giancarlo and Marisa. You mean a lot to me and I would like to share this important moment with you as well. Whatever happens in life, some people are and will always be important. I will always keep the beautiful years spent together as precious memories in my heart.

My family is so important and special that is almost impossible to put in words what they mean to me. First of all, I want to thank my uncles, aunts, and cousins. Especially, I would like to thank zia Margherita and zio Felice, who have always been special to me. I am also very grateful to Daniela and Giuseppe, how could I have done without the strudel and speck from Auronzo in these years! And my special thought goes to nonna Fabia, who is not here anymore: in my mind, I picture you during my defense, sitting in the front and with your hair fresh from the hairdresser, your beautiful smile, and absolutely no clue of what is going on around you…but incredibly proud and happy! Nonna Fabia, thanks for showing me the way every day of my life.

My sisters Claudia and Marzia are amazing persons, as well as my brothers in law Mauro and Massimiliano. And my little nephew Jacopo…he has started a new generation in my family bringing to all of us infinite happiness. Claudia, you are an example of loving kindness, courage, strength and dedication to people and their problems. You are great, humanly and professionally. During the tough time of writing this thesis, you told me how frustrating was for you to be far away and not helpul to me. Well, I can tell you that with your beautiful words not only you were helpful, but essential for me to carry on and not give up. I felt your support and love in every moment and I am extremely grateful for that. Marzia, you also supported me in an incredible way during these years. Especially in the darkest period, you were there to call or text me every day, checking how how was feeling and encouraging me to go on. Also, the time I spent at your place in Auronzo with you and Jacopo helped me a lot to overcome the sad and difficult moment. I don't know what I would have done without you. I see what you do every day, with your students, Jacopo and all the people around you and I feel so proud, you are inspiration for my work and for my life. Jacopo, you are too little to understand the importance of this book for zia Laura, but it doesn't matter. I want you to know that I love you so much and when I look at you my heart is full of happiness! Mauro and Massimiliano, thank you for being part of our family, the time I spend with you is simply fantastic and with both of you I always feel home.

Finally, mamma e papà. People always ask me what I talk about with my parents every day on Skype. And my answer is “about my day”. I tell you how is my day at work, what I do in my spare time, who my friends are, and so on. You know everything and everybody in Enschede, and I am happy that I could share all this with you. You were my strength during these years: a lot of people come and go, but you are always there. I want to thank you for everything you have done for me in my life and dedicate this achievement to you. I know how hard these years were for you. I could see it everytime papà took me to the airport, looking at me leaving and trying to hide the tears. I could understand it everytime mamma complained, after our leaving, that the house was empty. I could feel it everytime we were talking on Skype and Ciccio could listen to my voice without understanding where it was coming from. But I also know that seeing me happy of my work and my life makes you happy too…all this wouldn’t be possible without your infinite love and support. Grazie infinite del vostro amore, di avermi insegnato il valore dei sentimenti genuini e degli affetti sinceri, di avermi trasmesso l’importanza della famiglia, di avermi dato la possibilità di studiare, di viaggiare e di inseguire i miei sogni. Vi voglio immensamente bene!

Laura M. Daniele
Enschede, the Netherlands, May 2011

Contents

Chapter 1

Introduction

This thesis proposes a *layered methodology* for the development of *context-aware mobile applications*, which are distributed applications that can provide advanced and personalised services to their users. In this methodology, we model the behaviour of context-aware mobile applications in several refinements steps, from abstract specifications to final implementations, by guaranteeing *behavioural correctness* throughout these steps. *Automated model transformations* are developed to guarantee this correctness and generate executable behaviour refinements that in principle can be implemented by using different target technologies. This chapter presents the motivation of this thesis, discusses the main research objectives and outlines the adopted approach.

This chapter is organised as follows: Section 1.1 provides some background that is relevant for our research, Section 1.2 motivates the work in this thesis, Section 1.3 outlines our main research objectives, Section 1.4 presents the approach adopted in this thesis, Section 1.5 describes the scope of this work, and finally Section 1.6 presents the structure of this thesis.

1.1 Background

Today's panorama of service offerings is characterised by the widespread diffusion of the Internet and Web-based technologies everywhere in society (business, government, health-care, entertainment, leisure, etc.). We are surrounded by devices that can support us in several tasks of our every-day life, for example, e-readers to have always access to books and magazines, or mobile phones with extremely intuitive user interfaces for browsing, checking emails, keeping in touch with colleagues and friends through social networks, finding maps and locations, and so on. Moreover, this daily experience would not be possible without ultra-fast networks and wireless technologies that allow us to exchange any kind of data, such as audio, video, etc., anywhere, in real time and at low-cost.

In this panorama, it becomes of vital importance for service providers, either huge global organizations or local small business, to offer services that are innovative and distinctive [1-3]. On one hand, service providers have to preserve current customers and also attract new ones in order to survive and prosper in an ever growing arena of competitors. On the other hand, service users become more and more aware of the opportunities offered by the continuously evolving technologies and, consequently, more demanding and with higher expectations than in the past.

These facts lead to a new paradigm that moves the centre of information and communication control from the providers to the users. Users expect a profusion of services wherever they are, whatever they are doing and according to their personal preferences and needs. Providers have to create a wide range of enriched services in a rapid, low-cost and user-centric way. According to this new paradigm, the services being offered by the providers have to fulfil some important additional requirements, which we briefly describe as follows:

- *Ubiquitous*: services should exist anywhere and at anytime. In other words, services should always be available to the user, who expects services to be accessible in any moment wherever he may be. This does not concern the functionality of the service, but the availability of the service to the user. Availability depends on many factors, such as the characteristics of the user's device and the network. Another term used in the literature with a similar connotation as ubiquitous is *pervasive*. Ubiquitous is defined as "being everywhere at the same time, omnipresent" [4], while pervasive as "to become spread throughout all parts of" [5].
- *Context-aware*: services should be able to sense the user's context and, in case of changes in this context, autonomously adapt their behaviour in order to satisfy the user's current needs or anticipate the user's intention. Context can refer to the user's device, the network connection being used, personal user's information (location, activity, health condition), or physical environment characteristics (temperature, humidity, light). As an example of a context-aware service, a user's device could sense when its user is sitting in a movie theatre and consequently mutes itself without explicit user's intervention. When the user is travelling and dinner time is approaching, the same context-aware device could provide another service that suggests a suitable restaurant based on the user's location and his previous dining history.
- *Mobile*: services should be provided to mobile users. This is enabled by the availability of increasingly powerful and versatile portable

communication devices. These devices rely on the use of wireless technologies and 3G networks, which allow users to be continuously connected to the Internet and experience the flexibility of the Web also on their mobile devices. In order to offer to the users this kind of experience, services should adapt as transparently as possible to changes in the capabilities of the communication infrastructure by, for example, switching to a higher quality network connection as soon as it becomes available.

– *Personalised*: services should allow the users to configure their own preferences. It is interesting to investigate to which extent the user expects to specify what the service should or should not do for each particular situation, or whether this task should require explicit user's intervention. Services should be able to enforce the user preferences, either reacting to explicit user requests or reasoning about the user's context and learning from the user's previous choices or behaviour.
– *Composable*: services should be composable, in that it should be possible to enrich the user's experience by creating services as compositions of other available services. An example of composite service in telecommunications is the combination of voice conferencing, messaging, secure interactive data access and location-based service. A composite service should be assembled from generic service building blocks offered by several partners and bundled together to form full-service offerings.

The characteristics mentioned above are meant to reinforce each other and, sometimes, can be overlapping. For example, ubiquity is related to the capability of the service to be available to the user anywhere and at anytime, while mobility is related to the capability of the user to access a service through mobile devices and, therefore, anywhere. Thus, ubiquity and mobility are sometimes perceived as the same requirement. However, we could have a ubiquitous service without device mobility by positioning fixed computers and devices anywhere in a (bounded) environment. In the rest of this thesis, we refer to ubiquitous, context-aware, mobile, personalised, and composable services simply as *context-aware mobile services*.

1.2 Motivation

In parallel to the increasing user's demand for innovative and distinctive services, we are also witnessing the disruption of the traditional discrete structure of the industry, in which companies used to focus on a well-defined fragment of the market in order to provide a specific service. Nowadays, new players can enter the service market and offer various services. In the telecom market, for example, newcomers can use the infrastructures of the traditional telecoms due to the governments enforced liberalization, and also benefit from new technology that made service development easier and faster. Moreover, the service's lifecycle has dramatically shortened. In the past, the average time from concept to delivery of new services was 12-18 months. We now talk about lifecycles of weeks [1, 6]. As a consequence, service providers, either existing companies or new competitors, have to introduce innovative and distinctive services rapidly and at low-cost to remain competitive in this emerging market.

A representative example of this new paradigm in service offerings is given by the telecommunications industry [1-2]. For decades, a few large telecommunications companies have monopolised the market and controlled the user's experience. Consequently, the telecommunications world was extremely static: on one hand, the user did not have strong requirements and was just waiting for the providers to offer a new service; on the other hand, the technology was not evolving as fast as nowadays, and telecom service providers did not have to introduce new services in the market that often. In this static situation, the introduction of a new service was slow and costly due to integration and interoperability issues in an infrastructure that was not really designed for changes. Recently, when new competitors started entering the market and users became more and more demanding, it was clear that the old rigid telecommunications world had to be replaced with a dynamic and flexible environment. Therefore, not only telecom service providers, but service providers in any application domain, are nowadays forced to tackle both the technological challenge arising from the user demand of advanced and personalised services, such as context-aware mobile services, and the business challenge of introducing new service offerings in a rapid, low-cost and flexible way.

In order to cope with this two-fold challenge, one should look both at the state of existing technologies that can support the development of advanced and personalised services, and practices that can speed up time-to-market and cut costs in the process of creating and deploying new services. Since the technology evolves extremely fast and is mature enough to tackle the technological challenge [7], the focus moves to the development practices that can provide support to the business challenge. Therefore, the question is how to use the existing technology to satisfy the user's demand for service offerings and, at the same time, facilitate the business to create these offerings. In order to do that, this work aims at making the development process of services easier, faster, and cheaper. We briefly identify here some requirements that such a service development process should fulfil:

– *Intuitiveness*: in order to allow everybody, eventually also the users themselves, to provide services, the service development process should be intuitively appealing. This means that the steps of the process should be easy to follow, the language(s) used to model the service should be appealing for the user's interaction (without loss of expressiveness), and the tools simple to use. Ideally, the design of the service should be realised by just selecting components as building blocks and tying them together.

- *Abstraction*: advanced service developers may want to go into the details of the services being developed. This can be done without loss of appeal to intuition by dividing the development process in several abstraction levels. In this way, the intuitively appealing development environment mentioned above can be suitable for novices as well for advanced users.
- *Correctness*: services should behave in the way they are intended to behave. Especially in an extremely competitive market of service offerings, correctness can speed up the process and reduce the costs of introducing new services. Systems that do not present undesirable behaviour can be integrated more straightforwardly with existing services than services that are not correctly specified. This integration also depends on other factors, such as, for example, the complexity of the services and the integration goal. Correctness of system behaviour should be guaranteed throughout the whole service development process, from abstract design models to concrete implementations. Possibly, behaviour correctness should be already assessed in early stages of the development process, e.g., by simulating the behaviour of the system under development before investing in its implementation.
- *Agility*: since speed to market is a key driver in today's service development, the workload for designing, developing and provisioning a service should be minimized to enable rapid development. This can be done through, for example, the systematic re-use of models, processes and code, and the automation of the development steps.
- *Design for change*: since the platforms on which services run are replaced when their technology becomes obsolete or evolves, the development process should anticipate the possibility of platform changes in the design phase. This issue can be addressed by separating the application functionality from the technology with which this functionality is realised.

Currently available approaches, such as Service Oriented Architecture (SOA) [8] and Model-Driven Architecture (MDA) [9] can be used to support a service development process that addresses the requirements mentioned above. For both these approaches, there are some benefits and limitations.

SOA aims at facilitating distributed systems design through the disciplined use of the service concept. As defined in [10], services can be described, published, discovered and dynamically assembled for developing massively distributed, interoperable and evolvable systems. The problem with SOA is that it is too general: it cannot be used alone in service development since it is just a specific architectural style that prescribes how to build architectures by using the service concept. One could argue that SOA is intentionally general in order to allow embedding it in a development approach of choice. Therefore, SOA needs to be embedded in a methodology that gives support to the whole service development process. This support can be provided by MDA guidelines.

MDA aims at facilitating distributed systems design through the separation of platform-independent (PIM) and platform-specific models (PSM) concerns, the systematic (re)use of different models at different abstractions levels, and the use of (automatic) model transformations [9]. The problem with MDA is that in the past much attention was given to structural aspects of the modelled applications, and less attention to the PIM level and the behaviour of these applications. As indirect evidence of this tendency, surveys [8] show that among the 21 diagrams offered by UML, the structural diagrams are heavily used, while behavioural diagrams are much less used. However, there is general consensus in MDA on the importance of behavioural models [11] and, recently, the number of MDA practioners that specify application behaviour at the PIM level is increasing.

Our work combines the benefits of SOA and MDA in the development of context-aware mobile applications and services.

1.3 Objectives

This thesis proposes a *layered methodology* based on *behaviour modelling and transformations* for the development of *context-aware mobile applications*, which are distributed applications that can provide advanced and personalised services to their users. The main objective of this thesis is to progress the state-of-the-art in model-driven development of context-aware mobile applications by taking into account the behaviour of these applications already in early stages of the development process. In order to achieve this, we refine the application behaviour in several steps, from abstract specifications to final implementations, and develop *automated model transformations* throughout these refinement steps to generate executable models and guarantee their behavioural correctness. Particularly, the thesis aims at providing support for the following aspects, which are schematically depicted in *Figure 1*.

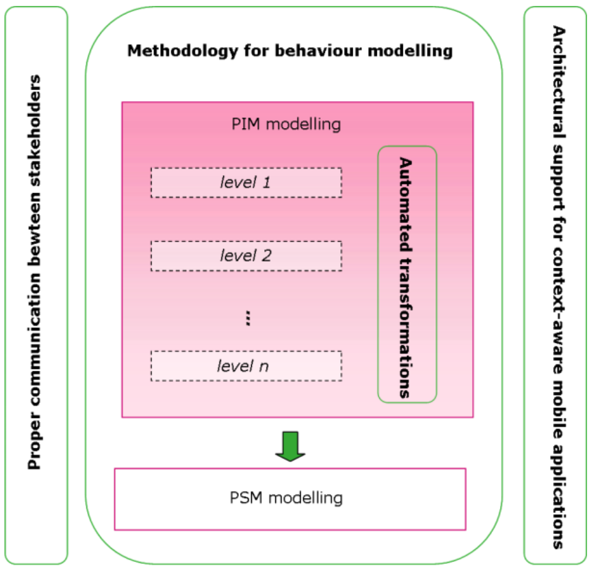

Proper communication bewteen stakeholders
Methodology for behaviour modelling
PIM modelling
level 1
level 2
⋮
level n
Automated transformations
PSM modelling
Architectural support for context-aware mobile applications

Figure 1 Objectives

Layered methodology for behaviour modelling
We aim at modelling the behaviour of services from abstract design towards implementation. In order to achive this, we propose a methodology for behaviour modelling that is decomposed in PIM and PSM modelling phases, as shown in *Figure 1*. This thesis focuses on the *PIM modelling* phase highlighted in *Figure 1*, which is decomposed in several abstraction levels that can be used to incrementally add technical details to the modelled application, assess the correctness of its behaviour during early stages of the development process, and verify whether the modelled behaviour conforms to the user requirements.

Proper communication between stakeholders
We envision a development process that promotes a common understanding among all the stakeholders, especially between business and IT experts. Our methodology aims at facilitating the development of advanced services from a technical perspective, i.e., the perspective of the services' developer, and also from the perspective of other stakeholders, such as business analysts, business managers, and even end-users. By providing a layered methodology that is intuitively appealing and possibly automated, we allow each stakeholder to address the (same) development process at the right abstraction level, i.e., high abstraction level for business people and lower abstraction level for technical developers. Therefore, we depicted the objective of proper communication among stakeholders in

Figure 1 as orthogonal to both the PIM and PSM behaviour modelling phases of our layered methodology.

Architectural support for context-aware mobile services

Our methodology is tailored to a specific category of advanced services, i.e., context-aware mobile services. For this purpose, we have defined a reference architecture that supports the general purpose functions that are commonly used by this family of services. This reference architecture influences both the PIM and PSM behaviour modelling phases of our methodology. Therefore, the architectural support for context-aware mobile applications is represented as an orthogonal objective with respect to these phases in *Figure 1*. However, this does not restrict our work only to this specific domain, since the knowledge developed in this thesis can be re-used with some adjustments in different application domains and with different reference architectures.

Automated support for PIM behaviour model transformations

Precise and unambiguous models and model transformations establish the first step towards automation. An objective of our work is to automate as much as possible the model transformations between PIM levels, as shown in *Figure 1*. In order to achieve this, the thesis investigates solutions to automate these PIM model transformations and implements some of these solutions.

1.4 Approach

Figure 2 depicts the research approach that we have adopted to define a layered and automated methodology for behaviour modelling and transformations of context-aware mobile applications. This research approach includes the following steps:

1. A *literature study* on general concepts and principles to be used throughout our research, such as the principles of MDA and SOA, and the general concepts related to the chosen application domain, namely context-awareness.
2. The *definition* of a *model-driven methodology* that:
   - separates the application development process in platform-independent and platform-specific modelling phases;
   - incorporates the application behaviour already at the platform-independent level;
   - decomposes the platform-independent level in several behavioural refinement transformations between models at different abstraction levels;

   – enforces reuse throughout these behavioural refinements by identifying patterns of recurrent behaviours, related to a reference architecture for context-aware mobile applications, in models at different abstraction levels.
3. A *survey* of *behaviour modelling techniques and languages* that support model-driven development, followed by an evaluation of their strengths and weaknesses, and the selection of suitable techniques that can be used for the purpose of this thesis.
4. *The identification* of the *ideal set of PIM models* for our layered methodology and *manual specification* of these models for a design example (called *Live Contacts*) on the realisation of context-aware mobile applications. The languages used for specifying these PIM models are selected based on the survey mentioned above.
5. The manual specification of *transformations between PIM models* in order to create systematic guidelines.
6. The *implementation* of *PIM transformations* based on the guidelines mentioned above using some transformation languages and tools.
7. The *application* of the proposed PIM modelling and transformations to a *PSM prototype* that uses some target technologies.

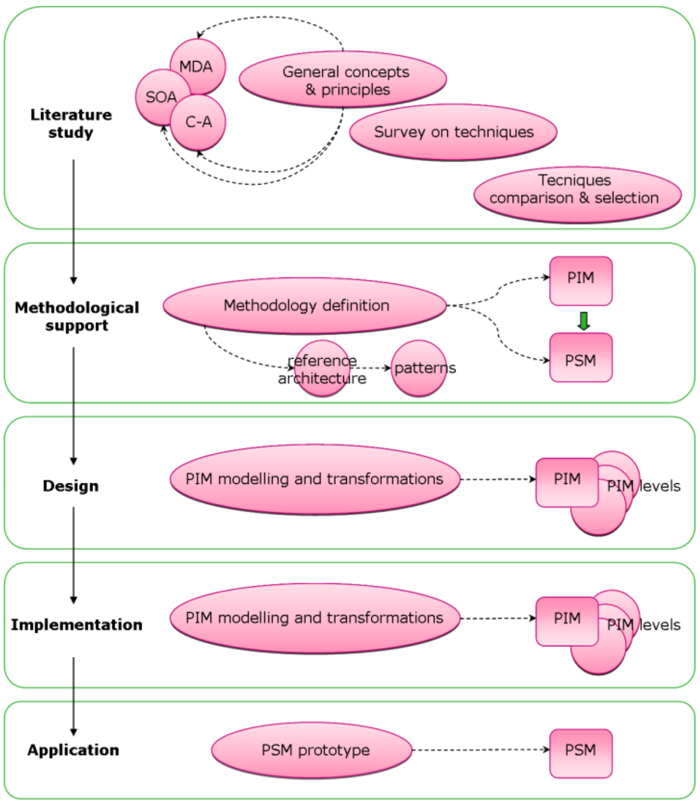


Figure 2 Our research approach

1.5 Scope and Non-Objectives

The scope of this thesis is the PIM behaviour modelling phase of a model-driven methodology for context-aware mobile applications, which is decomposed in consecutive behavioural refinements. The models generated by these refinements should be taken as input in the PSM design in order to be implemented with some specific technological solutions.

Our intention is to automate the proposed PIM behavioural refinements by using transformations. In order to achieve this automation, we define transformation rules based on interaction patterns that are related to a reference architecture for context-aware mobile applications.

In this thesis we do not extensively address the PSM design and neither do we automate the transformation from PIM to PSM. Moreover, the emphasis is not on the specific transformation languages and engines used to automate our transformation rules, but on the transformation rules themselves. We do not provide transformation rules and interaction patterns that support different reference architectures in addition to the reference architecture for context-aware mobile applications applied in this thesis.

1.6 Structure

The structure of the remaining of this thesis reflects the adopted research approach as follows:

- *Chapter 2 (General Concepts)* introduces the general concepts and terminology used throughout this thesis. The principles of MDA and SOA approaches, which can be beneficially applied in the development of distributed applications, are presented. Since context-awareness is the chosen application domain, we also discuss the basic definitions and related work in this domain.
- *Chapter 3 (A Model-Driven Methodology)* presents an overview of the MDA-based methodology we developed in our research and describes the SOA-based reference architecture we defined as part of this methodology. This methodology divides the PIM behaviour modelling phase in several levels with different degrees of abstraction. Each level is a refinement of the previous one and adds further technical details towards the implementation. This chapter identifies the ideal number of models and abstraction levels that should be used in our methodology, presents the general characteristics of these models, and discusses the transformations between these models. This chapter also introduces the concept of *interaction pattern*, which has been used to enforce reuse when automating model transformations.

- *Chapter 4 (Behaviour Modelling Techniques)* presents a survey of existing behaviour modelling techniques that can be used in model-driven development and discusses how these techniques can be positioned with respect to the abstraction levels of our methodology.
- *Chapter 5 (Techniques Comparison and Selection)* defines some qualitative evaluation criteria and compares the techniques presented in Chapter 4 with respect to these criteria. Based on this comparison, the chapter selects three solutions that are used in Chapters 6, 7 and 8, respectively.
- *Chapter 6 (Behaviour Refinement using A-MUSE DSL and ISDL)* discusses a solution that uses A-MUSE DSL and ISDL as modelling languages. This solution focuses on the behaviour refinement transformation of an abstract specification into an intermediate design model.
- *Chapter 7 (Behaviour Synthesis using Transition Systems)* discusses a solution that uses Transition Systems (TSs) to model behaviours. This solution focuses on the synthesis of the behaviour of an intermediate design model into a final design model.
- *Chapter 8 (Behaviour Refinement and Synthesis using BPMN)* discusses a solution that uses BPMN as modelling language. This solution focuses on both the behaviour refinement and synthesis transformations mentioned above.
- *Chapter 9 (Case Study)* applies the PIM refinement and synthesis transformations implemented in the thesis to generate a PSM level prototype, which is deployed on a BPEL engine and uses UDDI and web services technologies.
- *Chapter 10 (Conclusions)* presents our conclusions by stressing the main contributions of this thesis and identifying topics for further investigation.

Figure 3 depicts schematically the structure of this thesis.

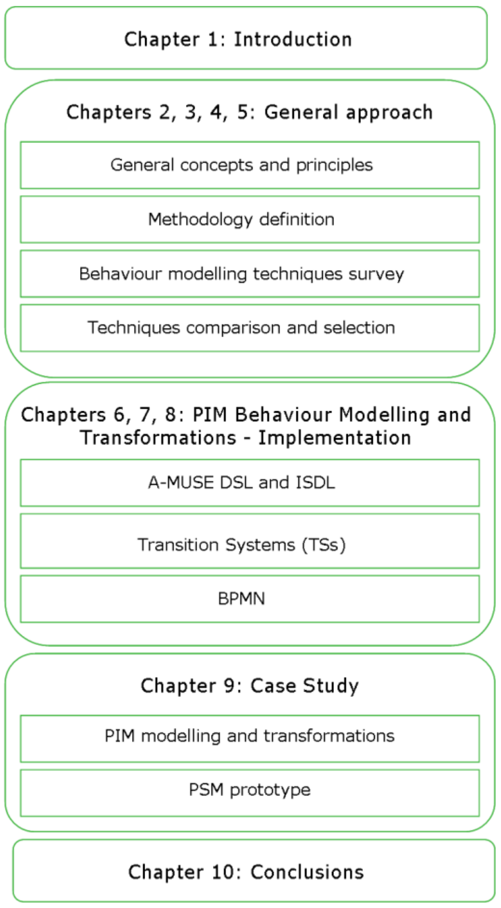

Chapter 1: Introduction
Chapters 2, 3, 4, 5: General approach
General concepts and principles
Methodology definition
Behaviour modelling techniques survey
Techniques comparison and selection
Chapters 6, 7, 8: PIM Behaviour Modelling and Transformations - Implementation
A-MUSE DSL and ISDL
Transition Systems (TSs)
BPMN
Chapter 9: Case Study
PIM modelling and transformations
PSM prototype
Chapter 10: Conclusions

Figure 3 Structure of the thesis

Chapter 2

General Concepts

This chapter introduces the general concepts and basic terminology used throughout this thesis. The principles of MDA and SOA approaches, which can be beneficially applied in the development of distributed applications, are presented. Since context-awareness is the application domain we have chosen in our research, we further introduce the basic definitions in this domain and discuss related work in this area.

This chapter is organised as follows: Section 2.1 provides some background on model-driven principles and concepts, Section 2.2 discusses service-oriented architectures, Section 2.3 introduces the basic notions underling context-awareness, and finally Section 2.4 describes the models for application development used in this thesis.

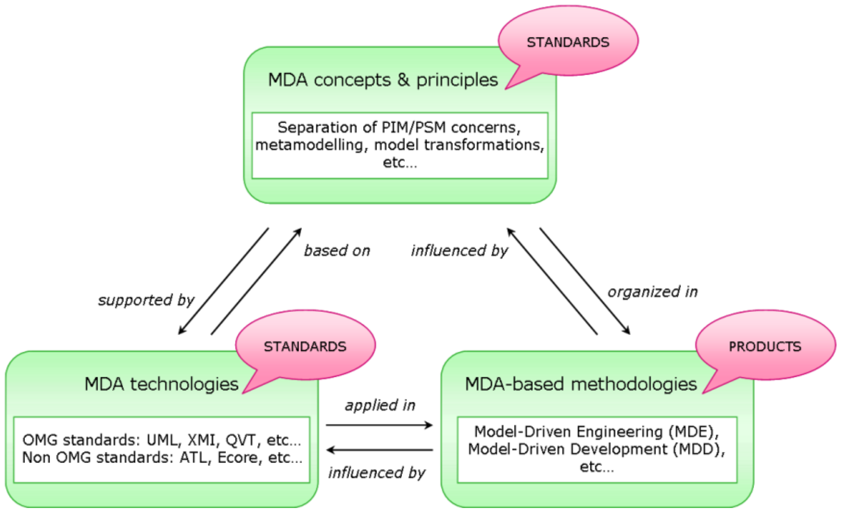
2.1 Model-Driven Architecture

Models help us handle the complexity in the everyday life: we use models to represent things, understand problems, communicate ideas, and memorise concepts. Analogously, in software engineering models provide a powerful and effective means to handle the complexity of the software life cycle [12-14]. By using models, software developers create abstractions of (parts of) the real system that needs to be developed (prescriptive models) or that has already been developed (descriptive models) [15]. In any case, these abstractions are useful to better understand the system, communicate with other stakeholders in a productive way, and make incremental improvements of the software product.

Some research areas in software engineering have explicitly recognized the role that models play in the software life cycle. Model-Driven Engineering (MDE) is based on the assumption that "everything is a model", in contrast to the basic principle of object technology that "everything is an object" [16-17]. Model-Driven Development (MDD) aims at developing models, rather than code, as the main artefacts of the software development process, in contrast to code-centric approaches in which systems are developed without using or maintaining intermediate models [18-19].

Model-Driven Architecture (MDA) [9, 20] is an initiative promoted by OMG (Object Management Group) [21] to support the realisation of the core MDE/MDD principles according to a set of standards. MDA provides a set of concepts and principles to guide the use of models in the development of distributed applications, and also the technologies that can be used to apply these concepts and principles to create real products. These concepts and principles, such as the separation of PIM and PSM concerns, metamodelling, and model transformations, are defined in an OMG standard [9]. Some technologies are also defined in OMG standards, such as, for example, UML [22], XMI [23], and QVT [24]. Some other technologies developed in the context of MDA are not OMG standards, such as ATL [25] and Ecore [26]. MDA concepts, principles, technologies and methodologies are represented in *Figure 4*, which shows that the MDA concepts and principles are supported by the MDA technologies, which are applied in the MDA-based methodologies, which are influenced by the MDA concepts and principles.

Figure 4 MDA overview


STANDARDS
MDA concepts & principles
Separation of PIM/PSM concerns, metamodelling, model transformations, etc...
based on
influenced by
supported by
organized in
STANDARDS
MDA technologies
OMG standards: UML, XMI, QVT, etc... Non OMG standards: ATL, Ecore, etc...
applied in
influenced by
PRODUCTS
MDA-based methodologies
Model-Driven Engineering (MDE), Model-Driven Development (MDD), etc...

Although MDA provides concepts, principles and technologies, it intentionally does not prescribe any particular development methodology. A development methodology consists of the set of activities, logical and temporal dependencies between these activities, roles that perform these activities and products that are artefacts of these activities, which are all involved in the development process of a distributed application [27]. Since

OMG did not want to impose a specific way to develop products on its members, vendors are left with the freedom to apply the MDA standards in combination with their own preferred methodologies. These MDA-based methodologies can be driven by several factors, i.e., business strategies, domain requirements, market demands, technical goals, and so forth. In any case, as depicted in *Figure 4*, these methodologies lead to products that are strongly influenced both by the MDA concepts and principles, and by the MDA technologies. Consistently with most of the literature, we use the terms MDE and MDD when the MDA concepts and principles are applied with an MDA-based methodology to the development of software (distributed) systems. There is some confusion in the literature and in the Internet about the precise meaning of these terms. This confusion is augmented by the use of more acronyms, such as Model-Driven Software Engineering (MDSE) and Model-Driven Software Development (MDSE) [28-30]. We refrain from discussing this terminology any further, since such a discussion falls out of the scope of this thesis.

In the following Sections we discuss the most relevant MDA principles and concepts used throughout this thesis, namely separation of concerns, metamodelling, model transformations, reuse, automation and execution.

2.1.1 Separation of concerns

Distributed applications should be developed according to a systematic process, which can help master the complexity of these applications, speed up their time-to-market, and decrease their development and maintenance costs [9]. As promoted by MDA, a way to achieve these results consists of separating the design of the following levels of models:

1. *computation independent models* (CIMs), which consist of business models that describe an application's requirements,
2. *platform-independent models* (PIMs), which consist of models that describe an application abstracting from the technological details related to the use of a specific technological platform, and
3. *platform-specific models* (PSMs), which consist of models that describe the application according to the technological platform chosen to implement the application.

The CIM requirements should be traceable to the PIM and PSM constructs that implement them, and vice-versa [9]. In other words, MDA promotes the design of the application's functionality and behaviour independent from the technology used to implement it. In this way, technology evolution does not affect the PIM design, which can still be reused with other specific platforms in other PSMs.

The notion of PIM and PSM is not absolute, but relative to the concept of platform itself. In order to refer to platform-independent or platform-specific concerns, one first needs to define what a platform is, i.e., which technological and engineering details are irrelevant in a particular context with respect to particular design goals [27, 31]. For example, for distributed applications a model can be considered a PIM when it does not prescribe a particular choice of middleware technology. Middleware technologies, such as, for example, CORBA [32] and Web Services [33] are infrastructures that facilitate the development of distributed applications by implementing common functionality that can be easily reused, and abstracting from implementation details, such as network technologies, programming languages, operating systems and hardware architectures. Therefore, a specific middleware technology can be considered as a platform to realise distributed applications. When a decision is made to use a particular middleware, the PIM model is transformed to a PSM model that uses the constructs of this middleware. However, this PSM model can be considered as a PIM, for example, with respect to the target operating system and hardware architecture.

2.1.2 Metamodelling

Another important MDA practice consists of representing the models created in the design by using some commonly agreed language in order to make these models available for all the stakeholders involved in the design process. Therefore, a requirement for models consists of promoting common understanding in enterprises between people with different skills, knowledge and background, such as business engineers and IT developers. Models are often used for discussion, communication and analysis, possibly not only within a single organization but also across organizations such as in the case of multi-organizational projects. Models are also used for design, validation, implementation, testing, management, and so forth. In any case, *models always have a purpose*. In this thesis, we start from the following definition of model:

Definition 1 Model

A model is an abstraction of a real world system defined using a notation that suits the purpose of the model.

The *syntax* is the notation used to represent a language. The *concrete syntax* is the symbolic notation of the language, while the *abstract syntax* is the conceptual notation of the language. *Figure 5* shows an example in UML, where the concrete syntax of the *class* element is represented by a rectangle with compartments, and the abstract syntax consists of a conceptualization of the *class* element itself, i.e., something which can have attributes and operations, and can be related to other classes by using associations, etc.

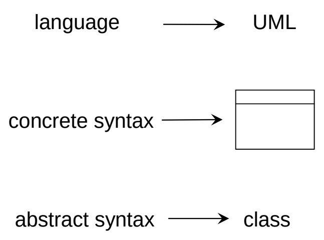


Figure 5 UML class example: concrete syntax versus abstract syntax

Definition 2 Metamodel

A metamodel is a particular type of model used to represent the abstract syntax of a language in a way that is machine readable and, therefore, can be manipulated by computerized tools.

The OMG has defined metamodels as "models of models". According to this general definition, we can have different types of metamodels depending on the purpose of the model that they describe. Although we embrace this definition, in this thesis we usually refer to the term metamodel as the specific type of model that describes the abstract syntax of a language. The OMG has also defined a standard language for expressing metamodels, which is the Meta Object Facility (MOF) [34]. The left part of *Figure 6* shows the relations between a model, the language used to represent this model, the concrete and abstract syntax of this language, and the metamodel that describes the abstract syntax of this language.

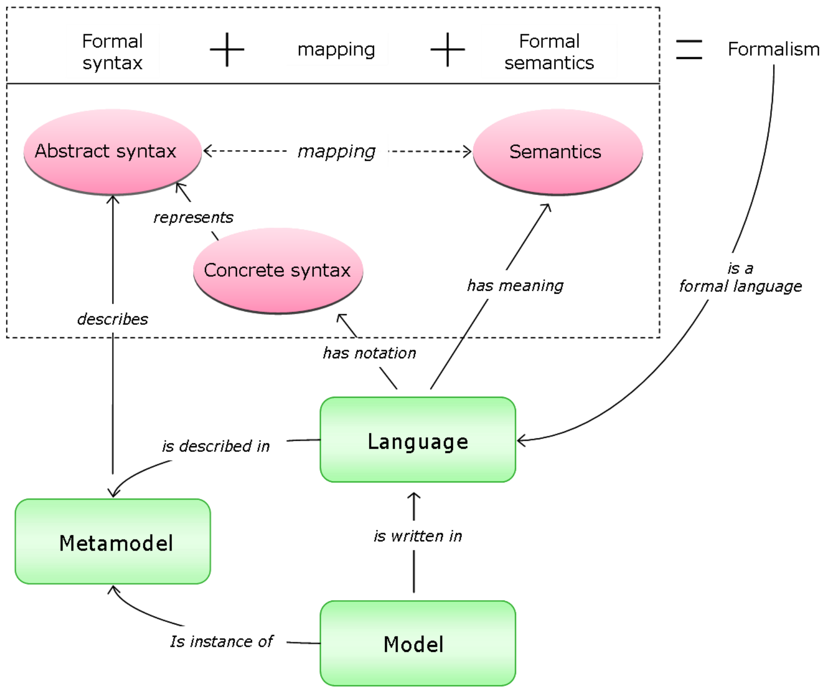


Figure 6 Models, languages, syntax, semantics and their relationships

Although necessary, it is not sufficient to have some syntax that represents the notational aspects of a language. As shown in the right part of *Figure 6*, a language should also have *semantics*, which describes the meaning of the language. For example, in the arithmetic language the syntactic symbol "+" is semantically associated with the operation of addition. The semantics can have an informal notation, for example, natural language. However, this makes the semantics ambiguous and can lead to misinterpretations. Therefore, it is advisable to have a formal notation for the semantics based on some mathematical theory, such as, for example, denotational semantics, operational semantics and so forth. When a language is endowed with precise and unambiguous, i.e., formal, definitions of syntax and semantics, we talk about formalism. Examples of formalisms are process algebras, Linear Transitions Systems (LTS), and Petri Nets (PN). Based on [35], we define formalisms as follows:

Definition 3 Formalism

A formalism F, or formal language, is a language consisting of a formal syntax L, a formal semantics S, and a mapping $M : L \longrightarrow S$ that relates the syntax L to the semantics S.

2.1.3 Model transformations

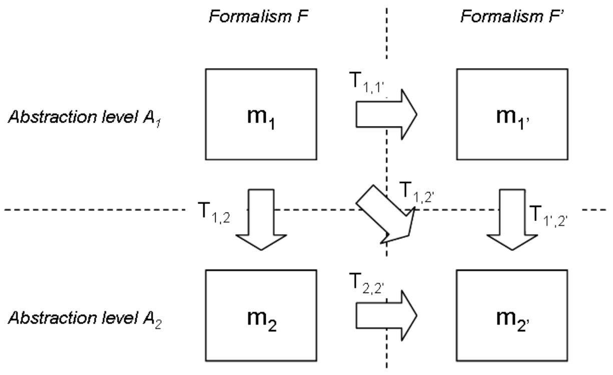
Model transformations play a central role in MDA approaches. Depending on the type of models that are involved, these transformations can have different purposes, which lead to different benefits. For example, one could be interested in transforming an abstract specification in a more detailed design model. In this case, the transformation is called *refinement*. One could also be interested in transforming a design model represented in some modelling language into an implementation model written in some programming language. In this case, the transformation is called *code generation*. In any case, a model transformation involves a source model and a target model.

Definition 4 Model transformation

A model transformation consists of the generation of a target model m_T (F') represented in a formalism F' from a source model m_S(F) represented in a formalism F.

When the formalism used to represent the source model is the same of the formalism used to represent the target model (F = F'), we talk about *endogenous* transformations. Vice-versa, when this formalism is different (F ≠ F'), we talk about *exogenous* transformations [15, 36]. A further distinction is based on the abstraction level of the source and target models. A transformation that converts between models at the same abstraction level is a *horizontal* transformation. In contrast, a transformation between models at different abstraction levels is a *vertical* transformation. While it is objective to evaluate endogenous versus exogenous transformations, it may be subjective to evaluate horizontal versus vertical transformations, since this latter evaluation relies on the ability of one to compute the abstraction level of the source and target models. *Figure 7* shows some examples of endogenous/exogenous and horizontal/vertical transformations, in which $m_S(F) \xrightarrow{T} m_T(F')$ denotes that $m_S(F)$ is the source model in a formalism F, $m_T(F')$ is the target model in a formalism F', and $\xrightarrow{T_{S,T}}$ is the transformation from source to target model.

Figure 7 Examples of model transformations



	VERTICAL	HORIZONTAL
ENDOGENOUS	$m_1(F) \xrightarrow{T_{1,2}} m_2(F)$ $m_{1'}(F') \xrightarrow{T_{1',2'}} m_{2'}(F')$ $A_1 \neq A_2$ Example: refinement	$m_1(F) \xrightarrow{T_{1,1'}} m_{1'}(F')$ $m_2(F) \xrightarrow{T_{2,2'}} m_{2'}(F')$ F = F' Example: refactoring
EXOGENOUS	$m_1(F) \xrightarrow{T_{1,2'}} m_{2'}(F')$ $F \neq F'$ Example: code generation	$m_1(F) \xrightarrow{T_{1,1'}} m_{1'}(F')$ $m_2(F) \xrightarrow{T_{2,2'}} m_{2'}(F')$ $F \neq F'$ Example: migration

The endogenous/exogenous and vertical/horizontal dimensions are orthogonal [15]. The transformations $T_{1,2}$ and $T_{1',2'}$ in *Figure 7* are endogenous vertical transformations. An example of this transformation is the model refinement mentioned above, which converts from an abstract model to a more detailed model in the same language. The transformation

$T_{1,2'}$ is an exogenous vertical transformation. The code generation mentioned above is an example of this transformation. The transformations $T_{1,1'}$ and $T_{2,2'}$ in *Figure* 7 are horizontal transformations. Moreover, when the formalisms F and F' are different, $T_{1,1'}$ and $T_{2,2'}$ are exogenous horizontal transformations. An example is the language *migration*, which converts a model in one language to an equivalent model in another language, e.g., for analysis purposes. In contrast, $T_{1,1'}$ and $T_{2,2'}$ are endogenous horizontal transformations when they convert between models based on the same formalism. An example is *refactoring*, which changes the internal structure of a model but not its external functional behaviour in order to improve some non functional aspects, e.g., readability or maintainability of the code.

A further classification is based on the way the transformation itself is defined. When one defines *what* the transformation does in terms of relations between elements of the source and target models, we talk about *descriptive* or *declarative* transformations. When one defines *how* the transformation is accomplished in terms of explicit steps, we talk about *prescriptive* or *imperative* transformations. Finally, transformations can be *manual* or *automatic*. Section 2.1.5 elaborates on the automation of model transformations.

2.1.4 Reuse

In order to increase the efficiency of the design process, both in terms of quality and costs, another important practice consists of collecting the knowledge acquired in some design steps and reusing it in other steps of the same design process and/or in the design of new applications, instead of creating these applications from scratch. In this way, it is possible to reuse best practices when creating families of applications, such as, for example, context-aware mobile applications in the case of this thesis.

The practice of collecting design knowledge during the design process in order to create reusable designs is called *design for reuse*. The practice of reusing existing design knowledge previously captured in other (steps of) design processes is called *design with reuse* [31]. In any case, reuse is possible at different levels in the design process, starting from models that capture core business processes and domain concepts, to code that implements specific designs solutions. Using the same principles as applied by manufacturers of hardware products, software product lines [37] can be created, illustrating reuse through a shared set of software assets and using a common means of production.

Since model transformations are essential in any model-driven development process, one should capture these transformations explicitly and reuse them consistently across solutions. Especially because defining a transformation is a time consuming task, which sometimes requires specialized knowledge of the application domain and the implementation technologies. In this way, it is possible to define standard transformations that make use of recurring patterns, which can be consistently applied, validated, and automated [38].

2.1.5 Automation

Models based on formal languages are machine readable and, therefore, constitute the basis for automation. Automation can be realised for several purposes, such as (1) analysis to check whether a model satisfies some desired properties, (2) validation to check models against requirements, (3) simulation to execute models in early stages of the development process, (4) transformation to generate more detailed models or executable code from abstract models, (5) testing to generate and execute test cases on the final implementation, (6) metadata management to handle relations between models and metadata, and so forth. A lot of effort in MDA is spent on the automation of model transformations.

Since model transformations can be applied in several steps in model-driven development processes, from initial analysis to code generation, the automation of these transformations can produce important benefits to the process, such as increasing the speed and enforcing the correctness of the implementations. *Figure 8* shows the standard approach used by MDA tools to automate model transformations. Elements of this approach are:

1. a *source metamodel*,
2. a *source model* instance of the source metamodel,
3. a *target metamodel*,
4. a *target model* instance of the target metamodel, and
5. a *transformation definition* instance of a transformation language.

A transformation engine, which is instructed with the transformation definition, takes as input the source metamodel, the target metamodel, and the source model. According to the transformation definition, this engine generates a target model, which conforms to the given target metamodel.

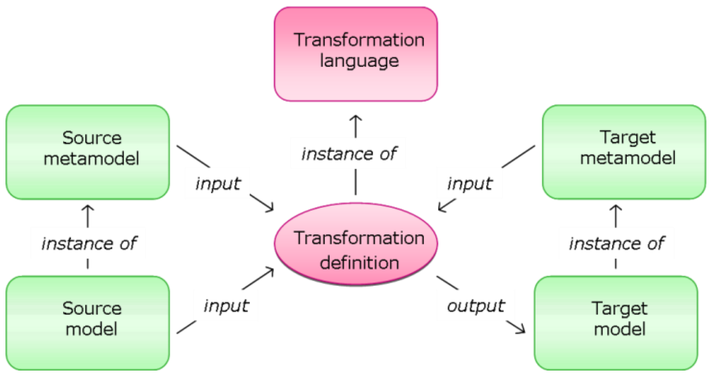


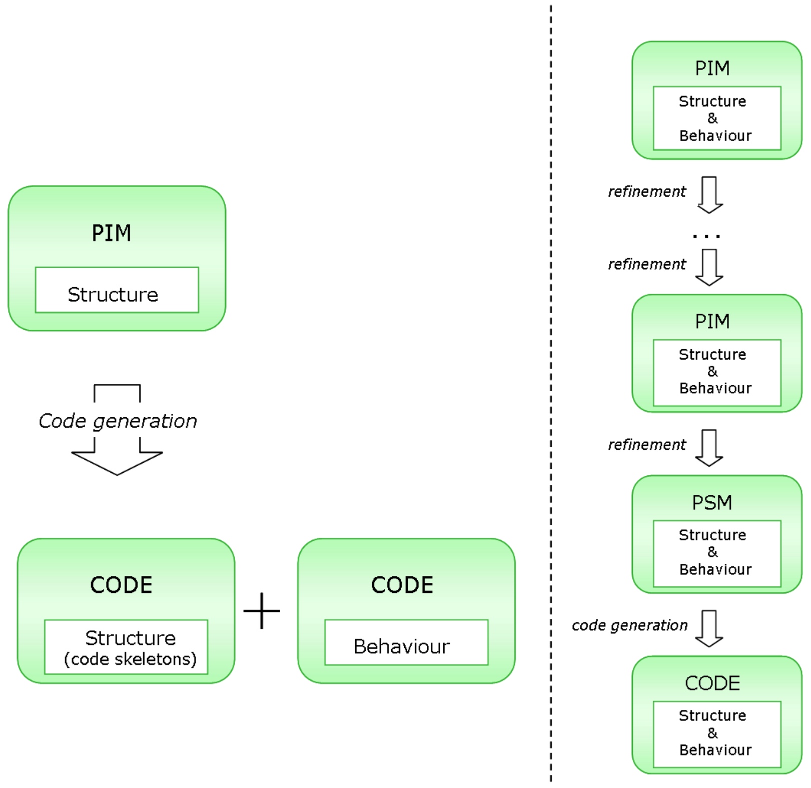
Figure 8 Automatic model transformations

OMG has defined a standard for model transformation languages called *Query/View/Transformation (QVT)* [24]. In order to handle declarative and/or imperative transformation definitions, the QVT standard defines different languages according to a layered architecture. In this architecture, *QVT Relations* and *Core* are declarative languages that allow the definition of transformations as declarations of relations among source and target metaelements. The so called *Operational Mappings* extends the Relations language with imperative OCL constructs. The Relations language has found the largest tool support, such as, for example, in the Medini QVT engine [39]. The ATL transformation language [25], which is inspired by the QVT standard, is a hybrid language that provides a mix of declarative and imperative constructs. The ATL language is supported by an ATL engine.

2.1.6 Behaviour Modelling and Execution

Considerable effort has been spent by the MDA community to model transformations meant to support application development [11]. However, as shown in the left of *Figure 9*, most of these transformations convert PIMs that describes an application structure into PSMs that implement this structure as code skeletons in some programming language. In this way, application behaviour is not defined at the PIM level and has to be incorporated at the PSM level by adding ad hoc hand-written code to the final implementation. Preferably, application behaviour should be incorporated to the application structure already at the PIM level of the design process and possibly refined in several PIMs that gradually add technical details to the design before generating the final code. This is shown in the right part of *Figure 9*.

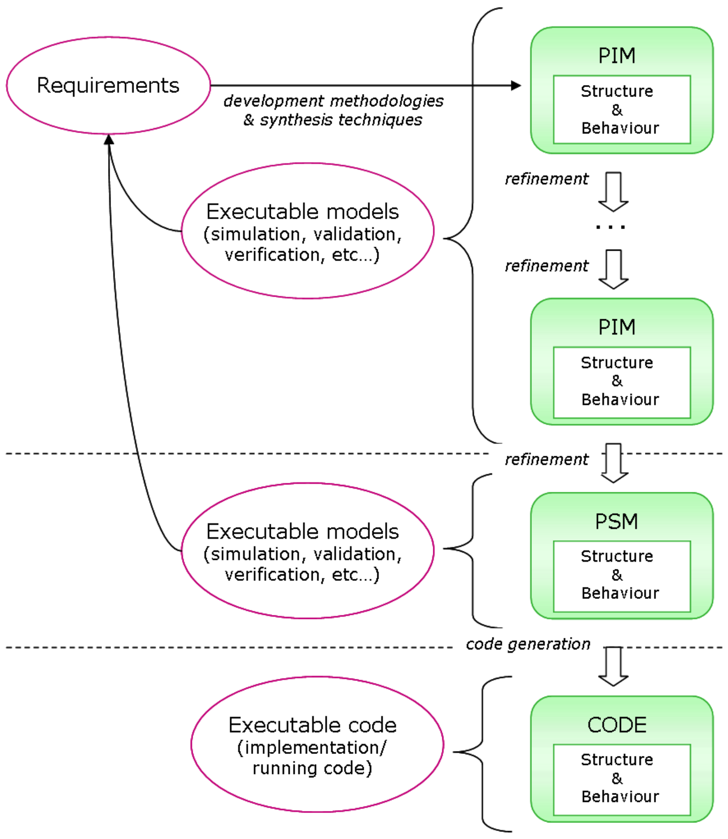
Figure 9 Behaviour modelling



Although there is general agreement in the MDA community about the importance of behaviour modelling, there is a lack of a commonly accepted modelling language to adequately represent behaviour [11]. For example, UML is a widespread standard that allows the representation of behaviours as sequence diagrams, statecharts and activity diagrams. However, UML lacks a formal semantics [35, 40], which is an essential part of a language as previously explained in Section 2.1.2. This issue is further discussed in Chapter 4, which is dedicated to the state-of-the-art in behaviour modelling languages and techniques. The investigation of such languages and techniques is the starting point of our research towards the development of a methodology for behaviour modelling.

Model execution at the PIM level is a desirable feature in an MDA development process. When talking about generation of executable models, we usually refer to the executable code at the PSM level that constitutes the final implementation of the application. However, when developing complex applications, it is advisable to have executable models in early stages of the development process before investing extensively in implementation [35]. Therefore, as shown in *Figure 10*, one should have executable models already at the PIM level before generating executable code at the PSM level.

Figure 10 Behaviour execution


Requirements
development methodologies & synthesis techniques
PIM
Structure & Behaviour
refinement
...
refinement
PIM
Structure & Behaviour
Executable models (simulation, validation, verification, etc...)
refinement
PSM
Structure & Behaviour
Executable models (simulation, validation, verification, etc...)
code generation
CODE
Structure & Behaviour
Executable code (implementation/ running code)

PIM models can be executed in different ways such as, for example, test and debug to check whether there are errors in our models, simulation of application behaviour, validation of this behaviour against requirements, verification of syntactical correctness in models at different abstraction levels, and so forth. Basically, when executing models at the PIM level we want to make sure that our application behaves in the way we have designed it before moving to the next development step. Chapter 4 further elaborates on behaviour modelling languages that can be used to create executable models at the PIM level.

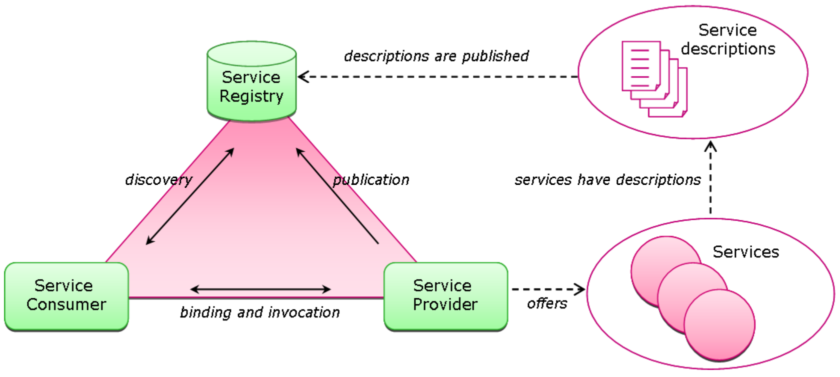
2.2 Service-Oriented Architecture

Service-orientation is a paradigm for the development of massively distributed, interoperable, and evolvable systems that considers services as fundamental elements, or building blocks [41]. Services are self-contained and modular components that can be described, published and discovered by using general agreed mechanisms [42]. Service-Oriented Architecture (SOA) [10, 43] is an architectural style based on the service-orientation principles, such as, for example, reuse of application functionality (reusability), abstraction from the way this functionality is implemented (abstraction), minimisation of interdependencies between units of functionality (loose coupling).

Figure 11 depicts a typical scenario in service-oriented architectures [42]. In this scenario, services are offered by *service providers* and are described in *service descriptions*. These descriptions abstract from implementation details and are used to advertise the service capabilities, interfaces, behaviour and quality. Service providers can publish the description of their services in a *service registry*. In this way, services are available for discovery to *service consumers*, who can consequently select and invoke the desired service based on the information of the description and without being aware of the details of the service implementation.

By prescribing the use of service interfaces as the only information necessary to communicate, SOA provides a means to abstract from implementation issues and to build application functionality independent from the specific technology used to realise this functionality. This also provides a means to generalise service logic and make it available for reuse, not only within an organization but also across multiple organizations, instead of rebuilding dedicated functionality every time.

Figure 11 SOA overview



Before the advent of service-orientation, distributed applications were often built in a dedicated fashion to address requirements one at a time [43]. In this way, the application only needed to fulfil a limited set of requirements and could benefit from the latest technology advancements. However, when significant changes were necessary due to new user demands or technology evolution, the solution was to build new applications from scratch. This led to redundancy of the implemented functionality, and consequently in development effort and expenses. Moreover, this resulted in interoperability issues among applications built with different technologies. The left part of *Figure 12* shows three (parts of) applications built with different technologies T_1, T_2 and T_3. The integration of these applications is complicated since each pair of incompatible technologies has to be connected trough a dedicated gateway, i.e., G_{12}, G_{13} and G_{13} in *Figure 12*. This way creates a huge amount of interconnections, which is not desirable. In contrast, by wrapping at design time each application in a commonly agreed description that abstracts from any specific technology, i.e., W_1, W_2 and W_3 in *Figure 12*, interoperability issues are definitely reduced. This is the context in which service-orientation was conceived.

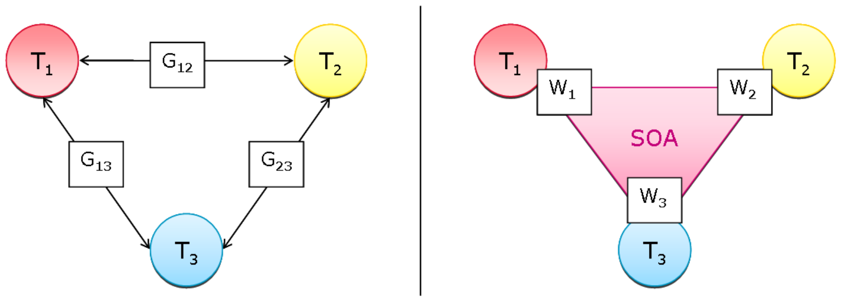


Figure 12 Service interoperability: dedicated applications versus SOA-based applications

Service-orientation is not a completely new paradigm in the IT history, since it incorporates elements from past paradigms. For example, SOA principles like service reusability, service abstraction and service composability, have been inspired by object-orientation. In a sense, SOA represents an evolutionary step, which shifts the application of modularity from a small scale to the potential modularization of the enterprise [43].

Although SOA is an architectural style independent from specific implementations, SOA is usually associated and sometimes equated to Web services [33]. The Web services technology has influenced service-orientation principles and a lot of vendors have implemented their SOA solutions by using this technology. However, SOA can be realised with any other technology for distributed systems, such as, for example, CORBA [32], Java RMI/EJB [44] and Jini [45].

In the following Sections we discuss the most relevant SOA principles used throughout this thesis.

2.2.1 Service orientation principles

Standardised descriptions

In order for services to communicate by making use of each other's functionality independently of the underlying implementation, it is necessary to describe these services in a standard way, i.e., in a way that can be unambiguously understood by all involved parties. Standardised descriptions (or contracts) state the purpose and the expected result of a service, its inputs, outputs, exchanged message types, operations, and address, and also service quality attributes, such as costs, performance, security, availability, and so forth [10]. In this way, service consumers know where and when to find a specific service, how to request and access it, and what level of Quality of Service (QoS) is guaranteed [46].

Loose coupling

Coupling measures the degree of interdependence between two entities, and loose (minimised) coupling makes these two entities as little as possible dependent on each others. In this way, each entity is defined as maximally self-contained, with simple message exchange patterns to allow access to the functionality exposed to the outside world (service interface). Therefore, loose coupling is a desirable property. Loose coupling can be achieved by eliminating unnecessary dependencies between entities and reducing the number of necessary dependencies [47].

Abstraction

The abstraction principle aims at hiding as much as possible the underlying details of a service and exposing to the external environment only the essential information necessary to make use of this service. This principle is tied to the fact that services in SOA make sense only from the perspective of what they provide or use without any mentioning of the internal details of how the service itself is implemented. In this way, a service can maintain awareness of other services abstracting from their irrelevant details.

Reusability

Design for reuse guarantees that services can realise some recurring functionality in multiple contexts of use within a single application or across a family of applications. A way to achieve reuse in the design consists of identifying common behaviour in different parts of the system and generalise this behaviour as services to make it usable by other (parts of) systems.

Composability
Services built as self-contained units of logic can be considered as building blocks for service composition. In principle, if one service solves a well-delimited problem, several services can be composed to solve a more complex problem. Service composition can take place at design-time or runtime. At design time, it allows the service developer to create services from existing ones (reuse of functionality), while at run-time it supports the on-demand request of end-users for personalised services. The services resulting from compositions may be used directly by service consumers, or can be also used by service aggregators as building blocks in further service compositions [10].

Interoperability
When using a specific technology to implement SOA, for example, web services, the principles mentioned above all support interoperability, although from different perspectives. Standardised service descriptions provide commonly agreed means to interoperate. Loose coupling promotes independency of services. Abstraction from implementation details reduces integration issues due to technology evolution. Reusability provides a means to share knowledge and functionality in a systematic way. Finally, composability uses services as first class elements to achieve complex common goals.

2.2.2 Concept of service

In the scope of this thesis we consider different perspectives of the service concept. These perspectives depend on the abstraction level at which we consider the system that we want to develop.

Figure 13 depicts a system and its environment. The environment consists of other systems capable to interact with our system, i.e., end-users or other applications. To simplify the discussion without loss of clarity, we assume that the environment coincides with the user. A system can have different types of users, such as managers, analysts, designers, developers, end-users, and so forth. In *Figure 13* we consider the integrated perspective of the system [48], i.e., regardless of its internal structure, and we define a service as "the external observable behaviour provided by the system as a whole". In other words, we are only interested in the service provided by the system to the environment, but not in how this service is provided. From this integrated perspective, there is a duality between the concepts of application and service. The application is the software system that offers a service to its environment (i.e., *what the system is*), and the service is the offered functionality (i.e., *what the system does*). Although these are two different concepts, at this level we can use them interchangeably without loss of clarity. For example, this is the case of context-aware mobile applications, which offer context-aware mobile services to their users.

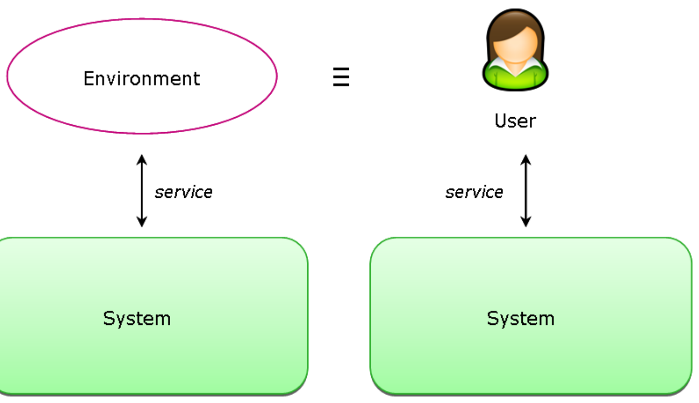


Figure 13 Integrated perspective: service as the observable behaviour of a system

In *Figure 14* we consider the distributed perspective of the system [48], in which we are not only interested in the service provided by the system to the environment, but also in how this service is provided. Therefore, we consider the system as a composition of interacting parts, which we call *(application) components*. According to SOA, these components make use of each other's services to cooperate in order to support the goals of the application. In principle, each component in *Figure 14* can expose its service to the environment independently of the specific application for which it was developed.

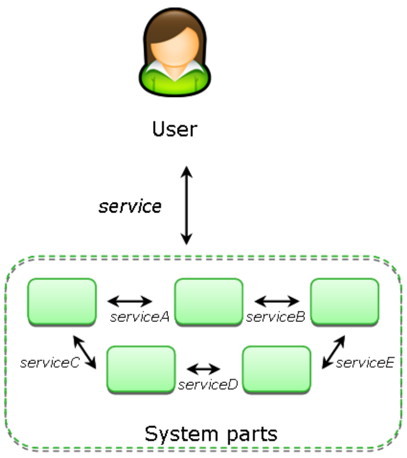


Figure 14 Distributed perspective: services as interactions among components to achieve the goals of the application

We can make the notion of system and service recursive by further decomposing each component and its service of *Figure 14* in more refined components. This decomposition usually stops when the final implementation of the system is reached.

2.2.3 Discussion

Service-orientation principles have been adopted extensively in both academia and industry. However, these principles themselves are not enough to build applications since they should be supported by a design approach. SOA does not prescribe such an approach. Consequently, some vendors, especially when facing budget and time constraints, build directly implementations without proper application modelling [43]. This introduces the risk of generating accidental behaviour, namely something that the application is not expected to do. Since this accidental behaviour can be detected only at run-time, changes have to be made on the implementations, which can be troublesome. In contrast, with a proper application design, accidental behaviour can be avoided or at least reduced. In order to exploit the full potential of service-orientation, SOA principles should be incorporated in an appropriate design approach. This motivates the use of SOA in this thesis in combination with an MDA-based design methodology, which can provide the missing methodological support in service-orientation when building architectures based on services.

2.3 Context and Context-Awareness

In this thesis we start from the definition of context given in [49], which is the following:

Definition 5 Context

Context is a collection of interrelated conditions in which something exists or occurs.

This definition implies that we always consider context as a set of conditions associated with a subject, which is something that "exists or occurs". In our models, a subject of context is called an *entity*. Although the concepts of entity and context are strongly tied, these concepts are fundamentally different. Actually, context is what can be said about an entity in its environment, which implies that context does not exist by itself [50]. The context of an entity is characterised by a "collection of interrelated conditions", which we call *context conditions*. Considering the context of a person (entity), examples of these conditions are the person's geographical location, conditions of the person's physical environment, such as temperature, humidity, light, etc., or the person's vital signs, like heart beat and blood pressure. Together, these context conditions form the person's context.

The context conditions mentioned above refer to real world phenomena that cannot be directly manipulated and used by applications as digital information. Therefore, it is necessary to represent these context conditions in terms of information that can be handled and interpreted by applications. We call this information *context information* and we define it by rephrasing the most referred definition of context in the context-awareness literature [51] in following the way:

Definition 6 Context information

Context information is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves.

Context-aware applications are intelligent applications that can monitor the user's context and reason about this context in terms of context information. In case of changes in this context, these applications can consequently adapt their behaviour in order to satisfy the user's current needs (reactive behaviour) or anticipate the user's intentions (proactive behaviour). For example, a context-aware mobile phone could be able to know when its user is sitting in a movie theatre and consequently mutes itself without explicit user's intervention. When the user is travelling and dinner time is approaching, the same context-aware mobile phone could suggest a suitable restaurant based on the user's location and his/her previous dining history. Anywhere and anytime, context-aware mobile applications should be able to provide relevant services to their users. The services offered by context-aware applications are called *context-aware services*.

Since the design of context-aware applications relies on a variety of components that are distributed over the environment, we can define a context-aware application as follows [50]:

Definition 7 Context-aware application

A context-aware application is a distributed application whose behaviour is affected by its users' context.

Figure 15 summarises the concepts defined above and shows how these concepts are related to each other.

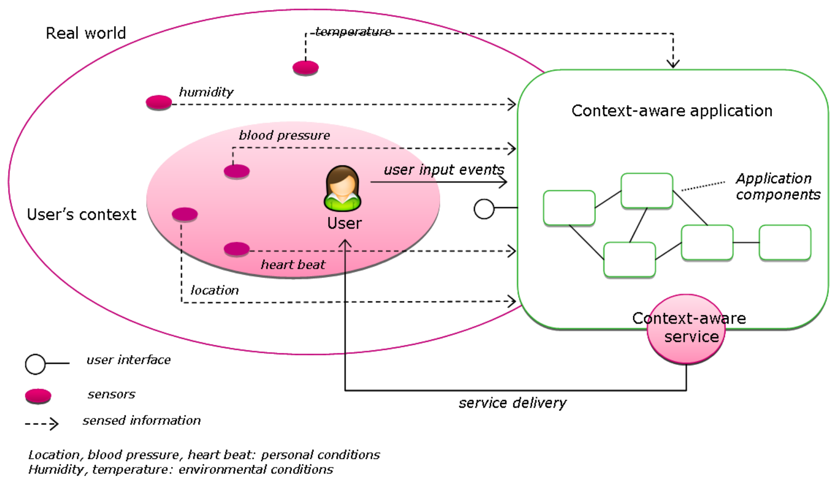


Figure 15 Context-aware applications and services overview

Figure 15 depicts a user in the real world. This user may provide some input events to a context-aware application through a graphical interface. This user has a context, which consists of personal conditions (e.g., location, blood pressure and heart beat) and environmental conditions (e.g., temperature and humidity). These context conditions are captured by sensors deployed in the user's environment, such as, for example, a GPS device integrated in the user mobile phone or a wearable device that can monitor the user's vital signs. Alternatively, some context conditions can be obtained from websites, such as, for example, locations from http://maps.google.com, and weather conditions from http://www.buienradar.nl. In any case, context conditions cannot be used directly as they are captured, but they need to be represented in a format that complies with an agreed context information model with specific values in order to be used as inputs by context-aware applications. The accuracy with which the context information values used by applications reflect the real context conditions captured by sensors or provided by web services is called *quality of context*. The more these values are close to the real context conditions, the better the quality of context is.

Figure 15 also shows that a context-aware application consists of application components, which interoperate in order to achieve some common goal. This common goal is realised in *Figure 15* by a context-aware service, which is delivered by the context-aware application to its user.

2.3.1 Context-aware middleware platform

Because of stringent time-to-market requirements, it is not feasible to build dedicated context-aware applications for each user's preferences and needs, since in this case application development would be time consuming and costly. In contrast, with a proper middleware platform, generic functions can be made available and reused in various context-aware applications without developing them from scratch. Depending on the specific application to be developed, this middleware platform can be configured by implementing functions that are not worth generalizing, since they are too specific to be used by other applications. As depicted in *Figure 16*, on top of this middleware platform, a context-aware application should also offer specific functions implemented by application-specific components.

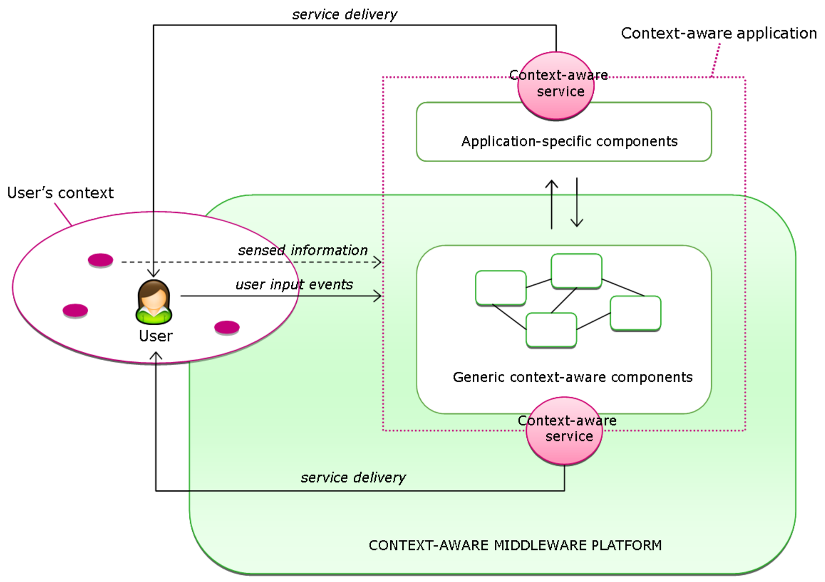


Figure 16 Context-aware middleware platform

According to the most relevant literature in the domain of context-aware applications [50, 52-54], the following three main aspects can be generalised in our middleware platform:

1. *context*, which concerns context gathering issues, such as retrieving context conditions from sensors or web services and eventually aggregating these values in higher-level context information;
2. *reaction*, which concerns the execution and delivery of services as reactions to context changes in the user context or user input events;
3. *logic*, which concerns the application behaviour that controls the aspects mentioned above.

Context

Context information can be raw information collected by sensors, e.g., location information from a GPS device, or aggregated information collected by several sources, e.g., user's activity information (for example, 'in a meeting' activity inferred from GPS locations and proximity with other people). In order to provide raw and aggregated context information, the sources of context information, which we call *context sources*, should be hierarchically organised.

Some work in the literature has dealt with such a hierarchical organization. For example, [51] introduced the concept of widgets, which encapsulate sensors that capture raw context information, and the concepts of interpreter and aggregator, respectively, to interpret context information and infer information from several widgets. Similarly, [55] proposed a context gathering layer consisting of sensors, interpreters and aggregators. In [54] the concepts of context provider and synthesizer are introduced. In this thesis, we have adopted the internal hierarchy proposed in [50] based on context sources and managers, which is depicted in *Figure 17*.

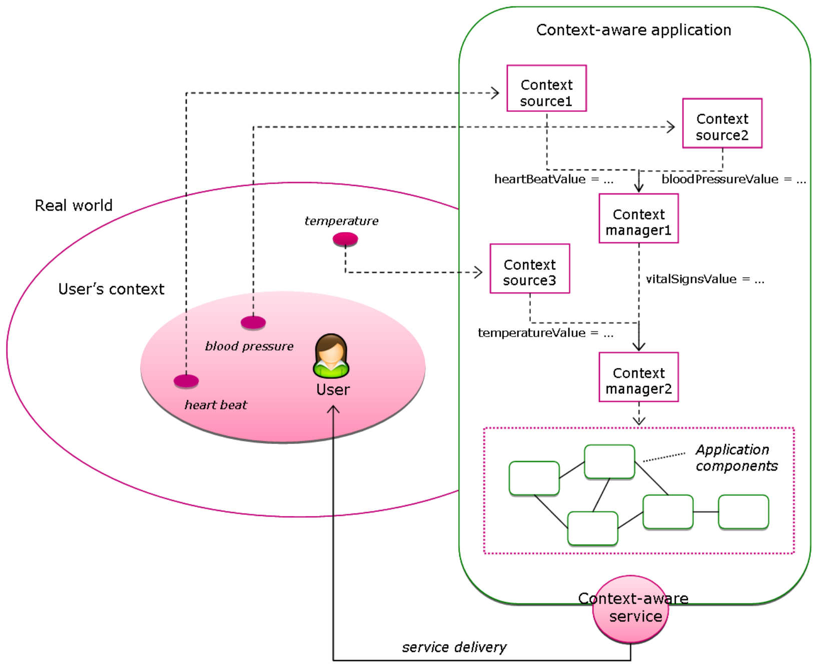


Figure 17 Context aspects

The context sources in *Figure 17* encapsulate single domain sensors, such as a wearable device that monitors the user's blood pressure. Context managers are able to combine the context information acquired by multiple domain context sources, such as the user's blood pressure and heart beat, to obtain aggregated context-information, such as the user vital signs in *Figure 17*. The hierarchy of context information processing is recursive in the sense that the outcome of context managers can become input to higher level context managers for further processing. For example, the user's vital signs, which are aggregated by the *Context manager1* in *Figure 17*, are then combined with the environmental temperature by the *Context manager2*. As a result of this hierarchy, a directed acyclic graph is created where the initial nodes are always context sources and the final nodes may be either context sources or managers.

There are two ways to provide context information to a context-aware application: *event-based*, i.e., when relevant changes occur in some context information attribute and these changes are notified by context sources or managers, and *query-based*, i.e., when some specific context information attribute becomes relevant at some point in time, context sources or managers can be directly queried to obtain the current value.

Reaction

Context-aware applications are characterised by reactive behaviours. An example of reactive behaviour is muting the user's mobile phone when the user is sitting in a movie theatre. Actions represent application reactions to context information changes, and these reactions may be the invocation of any service internal or external to the application, such as the generation of a signal, the delivery of a notification or a web services request. Since it can be beneficial for flexibility and reuse reasons to have a hierarchical organization also of reaction aspects, we have adopted the internal hierarchy proposed in [50] based on action resolvers, providers and implementors, which is depicted in *Figure 18*.

The action resolver component in *Figure 18* resolves compound actions into indivisible units of action purposes, such as *sendSMS* and *makecall*. These action purposes are defined by action provider components and describe an intention to perform an action with no indication on how and by whom. The responsibility of the action implementor components in *Figure 18* consists of defining various ways of implementing a given action purpose. For example, the action *makeCall* in *Figure 18* may have two different implementations supported by different telecom providers.

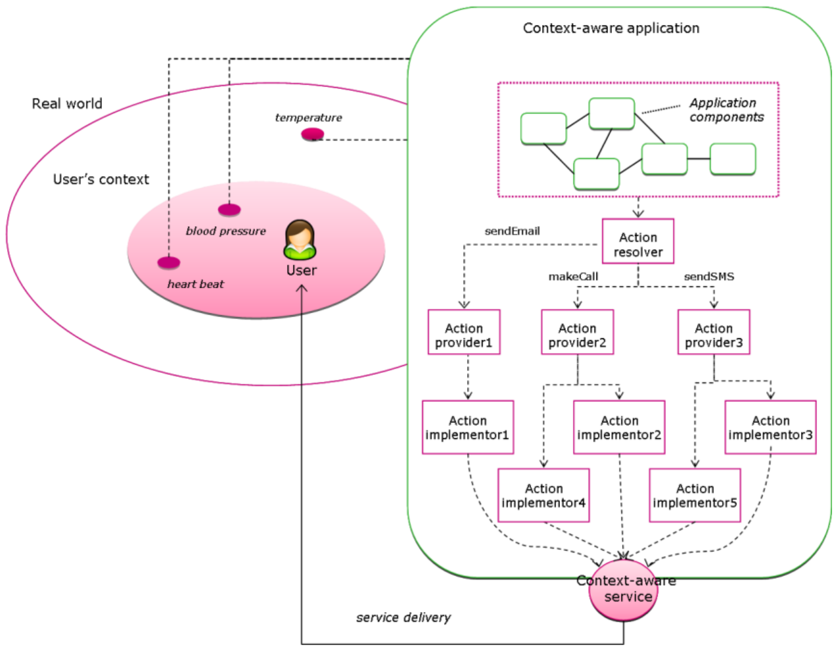


Figure 18 Reaction aspects

Logic
The logic aspect constitutes the core of a context-aware application since it determines how an application should behave upon the occurrence of context changes to generate proper actions. Therefore, the application logic is responsible for coordinating the context gathering and reaction execution aspects described above. In addition, the application logic is responsible for coordinating any other components, such as the discoverer component introduced in [51] to register the capability of context components, i.e., widgets, interpreters and aggregators, and action components, i.e., services that execute actions on behalf of the application.

Concerning some related work that has dealt with the application logic aspect in the context-awareness domain, in [51] an application is defined as the components that makes use of all the other components in the framework (widgets, interpreters, aggregators, services and discoverers). In [55] a layer of the proposed architecture is dedicated to implement the application logic, which organises the underlying layers dedicated to context management and query issues. In [54] applications consist of context consumers whose behaviour is designed by using rule-based approaches or learned by the applications themself by using some machine learning approach. In [50] the use of the Event Control Action (ECA) architectural pattern [56] is proposed, in which an *Event* module is dedicated to context concerns and to provide the application with events that model contextual changes in the application or its environment, a *Control* module is dedicated to observe events and to trigger actions as a consequence to these events, and an *Action* module is dedicated to perform the actions triggered by the control module. The control module constitutes the application logic and is implemented by using a rule-based approach.

How the application logic behaviour should be represented at a platform-independent level by using an appropriate modelling language, and realised with proper technologies at the platform-specific level by preserving behaviour correctness and consistency, is the main topic of this thesis. We aim at realising this application logic in a way that combines the benefits of MDA, which gives foundational methodological support to model application behaviour, and SOA, which gives architectural support to execute this behaviour.

Figure 19 shows an overview of the context, logic and reaction aspects as used in the remainder of this thesis.

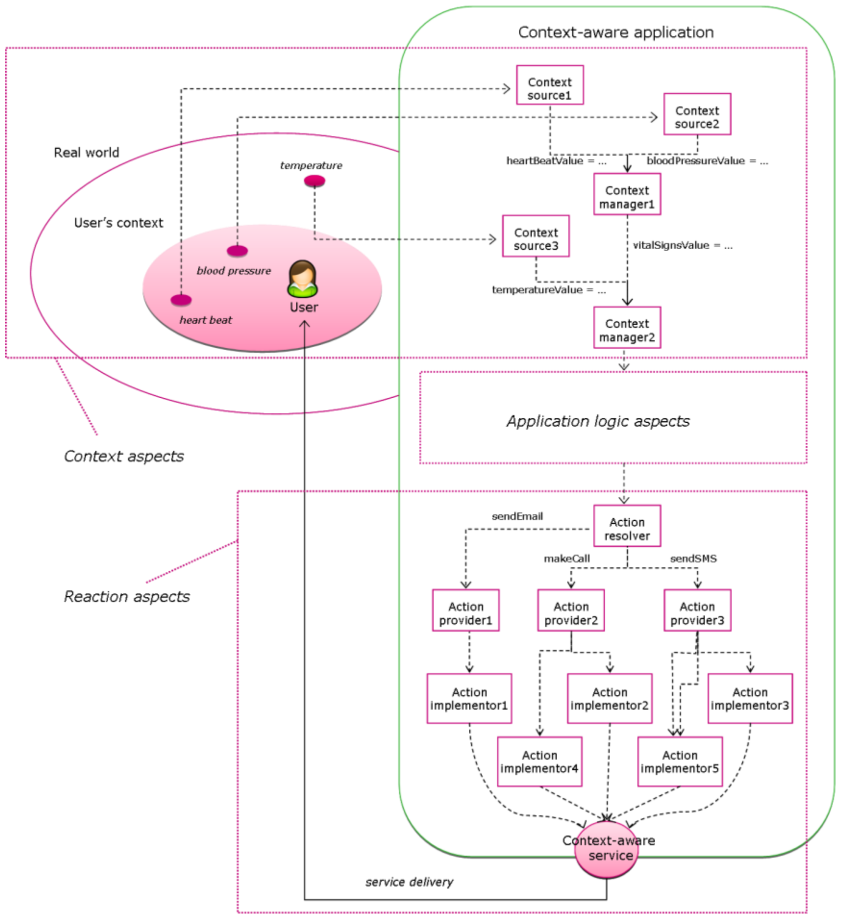


Figure 19 Context, logic and reaction aspects overview

2.3.2 Context modelling

In order for the components that manipulate context information to cooperate, a context-aware application needs to rely on a context model that defines the structure of this context information. Therefore, we define a context model that represents the relevant context conditions of entities in the application's universe of discourse. In the sense of [50, 57], we consider context models as conceptual models of context that represent a given subject domain in an abstract way, independent of specific design and technological choices. When we define a conceptual model of context, we abstract from any system design and technological detail, such as the way in which context is sensed, provided, processed and used.

Context models provide us with a conceptual foundation for the development process of context-aware mobile applications. Particularly, context models allow us to provide interoperability among application components distributed in the system for what concerns context information. Each of these components realises specific parts of the application logic and, in order to support the goals of the application, it has to interoperate with other components. Our context model should provide concepts (entities and context) that are commonly known and understandable. In other words, a context model is fundamental since it provides the common vocabulary to "make our components understand the same language". In this thesis, we propose context models that are based on [50], which provides foundational ontologies to support conceptual modelling and situation reasoning. *Figure 20* shows the foundation concepts used in our context models.

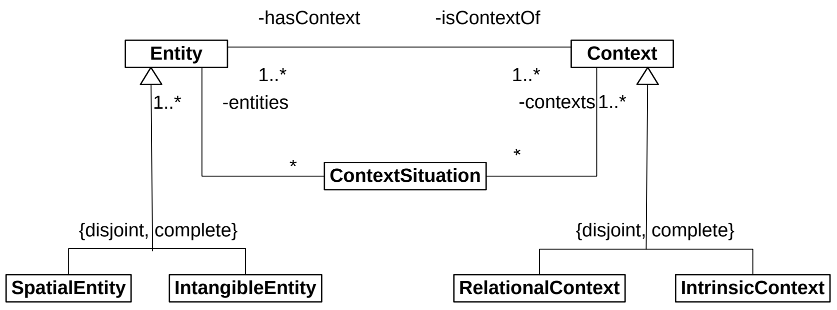


Figure 20 Context model

Figure 20 represents the Entity and Context classes. The term 'context' here corresponds to the context conditions mentioned in the previous Sections. Any entity may be related to several different types of context and a specific context type may be referred to one or more entities. For example, an entity Person can have Location and Activity context types depending on the physical position and the activity this person is engaged at a certain moment. In turn, the context type Location can refer to a Person entity as well as to other entities, such as a Device or a Building. *Figure 20* shows that the Entity class is further split in two other classes. The SpatialEntity class represents tangible objects, such as Person or Device, while the IntangibleEntity class represents intangible objects, such as Application or Network. The Context class is also split in two other classes. The IntrinsicContext represents a context type that belongs to the essential nature of a single entity and does not depend on the relationship with other entities. An example of intrinsic context is the location of a person or a device. In contrast, RelationalContext represents a context type that depends on the relation between distinct entities, such as the contact list of a user that relates a User entity to several Contact entities. Finally, *Figure 20* depicts the ContextSituation class, which relates contexts and entities. Context situations allow us to represent particular state-of-affairs of the applications' universe of discourse. For example, the context situation Proximity describes when a certain person is within a certain threshold distance from another person. Therefore, the proximity situation relates an entity Person to another entity Person through the context Location.

2.4 Models for Application Development

The following Sections discuss the type of models that we have used in this thesis for the development of context-aware mobile applications.

2.4.1 Information model

An information model represents the data handled by the application being modelled. In this thesis we use UML class diagrams to model these data in terms of classes, attributes, operations and relationships among these classes. *Figure 21* shows an example of an information model, which represents the classes User, Buddy and BuddyList, their attributes, e.g., the name of the user, their operations, e.g., getBuddyList() to retrieve the list of user's contacts, and their relationships, e.g., "the user has a buddy list" and "the buddy list is a set of buddies". The User, who is somebody registered in an application with a name, has a BuddyList, which represents all the contacts of the user in the application. The Buddy class provides detailed information about these contacts, such as their email, phone number and contact location. The application uses this information to offer its functionality to the user.

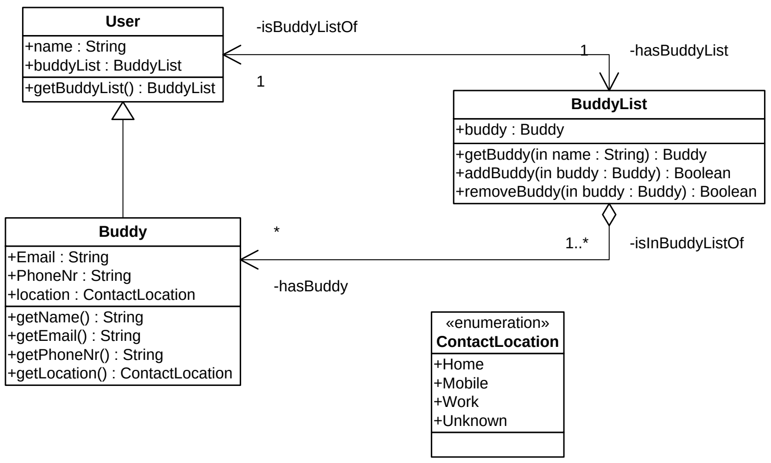


Figure 21 Example of information model

2.4.2 Context model

A context-model represents the relevant concepts used by application components that manipulates context. In this thesis we use UML class diagrams to model foundation concepts and instances of these concepts. *Figure 22* shows an example of context model complementary to the information model in *Figure 21*.

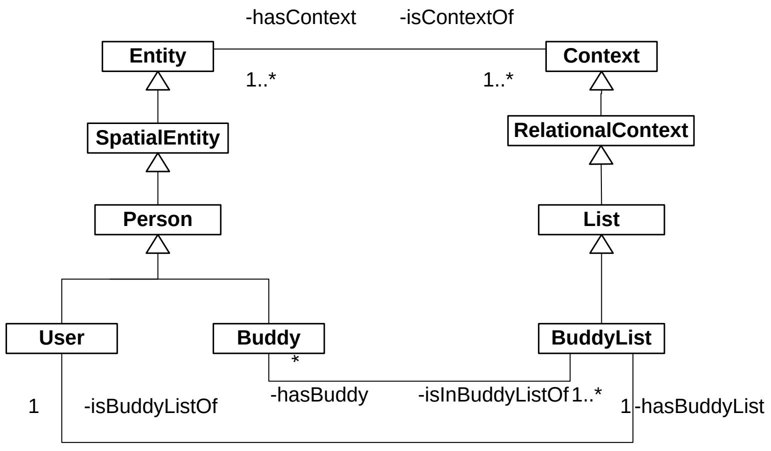


Figure 22 Example of context model

Figure 22 depicts the object Person, who is an example of SpatialEntity since it represents a tangible object. A List is an example of RelationalContext, since it represents a context type that depends on the relation between distinct entities. These entities are the User and the Buddy in *Figure 22*. The User has only one BuddyList, which is a collection of several Buddy objects. A Buddy can be in the BuddyList of one or several User objects.

2.4.3 Behavioural model

A behavioural model represents possible behaviour (sequence of activities) of the system to achieve the goals of the application. *Figure 23* shows an example of behavioural model as UML activity diagram. This diagram is based on the information model in *Figure 21*.

Figure 23 Example of behavioural model

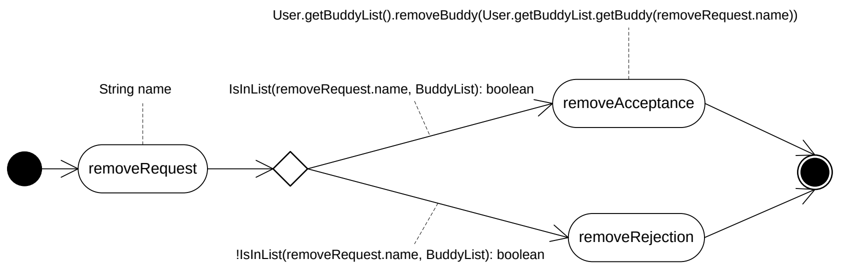

User.getBuddyList().removeBuddy(User.getBuddyList.getBuddy(removeRequest.name))
String name
IsInList(removeRequest.name, BuddyList): boolean
removeAcceptance
removeRequest
removeRejection
!IsInList(removeRequest.name, BuddyList): boolean

Figure 23 depicts an example scenario that allows a user to remove a buddy from his contact list. In this scenario, the removeRequest activity provides the name of the buddy to be removed. If this buddy is not in the user's contacts (!IsInList(removeRequest.name, BuddyList): boolean condition), the user request is rejected (removeRejection). Otherwise (IsInList(removeRequest.name, BuddyList): boolean condition) the request is accepted (removeAcceptance) and the buddy is removed.

Chapter 3

A Model-Driven Methodology

This chapter gives an overview of the MDA-based methodology we have defined for the development of context-aware mobile applications. We first present the basic elements of this methodology separately, such as our design models with different abstraction levels and the transformations between these models. We further introduce an essential concept of our methodology, i.e., the concept of *interaction pattern*, which has been used to enforce reuse during the automation of our model transformations. Since context-aware mobile applications are our target domain, we also present a reference architecture that can be used to develop such applications. We finally combine the basic elements, interaction patterns and reference architecture in a global methodology overview.

This chapter is organised as follows: Section 3.1 presents the basic elements of our MDA-based methodology, Section 3.2 discusses the approach based on reuse that we have taken to define the behaviour models and transformations proposed in this methodology, Section 3.3 presents the interaction pattern concept, Section 3.4 introduces our reference architecture tailored to context-aware mobile applications and services, and, finally, Section 3.5 gives the methodology overview.

3.1 Basic Elements

Since this thesis aims at providing a model-driven software development methodology, we start from the following definition inspired by [58-59]:

Definition 8 Software development methodology

A software development methodology defines in a structured and systematic way the set of activities, roles that perform these activities, products that are artefacts of these activities, and logical and temporal dependencies between them (activities, roles and artefacts), in the development process of software applications.

Since our methodology is model-driven, the artefacts consist of models, and the main activities consist of transformations between these models. *Figure 24*, which elaborates on *Figure 9*, shows these models and transformations, which are discussed in the following Sections together with the approach we have followed to realise them.

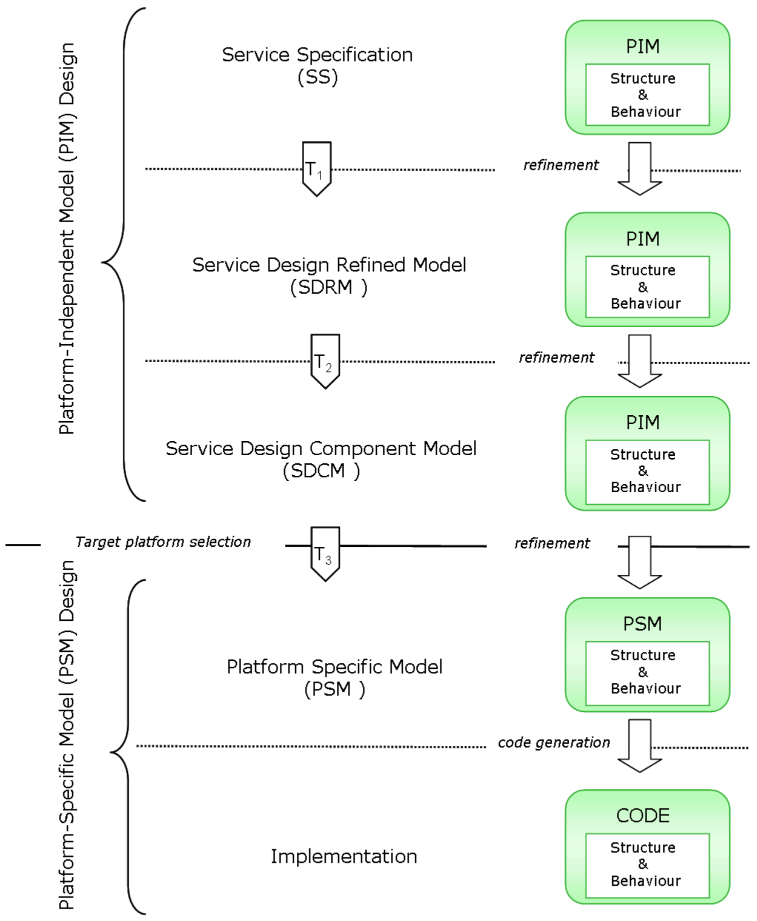

Platform-Independent Model (PIM) Design
Service Specification (SS)
PIM
Structure & Behaviour
T1
refinement
Service Design Refined Model (SDRM)
T2
Service Design Component Model (SDCM)
Target platform selection
T3
Platform-Specific Model (PSM) Design
Platform Specific Model (PSM)
PSM
code generation
Implementation
CODE

Figure 24 Overview of models and transformations at different abstraction levels

According to the MDA principle of separation of concerns, we divided our methodology in different levels of models with different degrees of platform-independence. As shown in *Figure 24*, we first divided the design process in PIM and PSM design phases. Since application behaviour should be already incorporated in the PIM design phase, we focused on reasoning at this level. We decomposed our PIM level in several models, where each consecutive PIM model adds technical details to the previous one. Initially, we modelled only two levels of platform-independence, namely SS and SDCM in *Figure 24*. Since the gap between SS and SDCM was rather wide, and correctness and consistency, particularly of behavioural aspects, were hard to guarantee in a single transformation step, we learned that an intermediate level (SDRM) was necessary. As a result, we recommend three levels, namely SS, SDRM and SDCM as the ideal set of PIM models that should be used in our methodology.

3.1.1 Models

The *service specification (SS)* is the most abstract model of the methodology and describes the application to be developed as a single entity with behaviour from an external perspective only. At this level, we specify the service that our application offers to its user and we do not consider any structural detail of the application, i.e., we do not have any knowledge yet about its internal components. As shown in *Figure 25*, the SS models the system as a black box, which receives some inputs from the environment and eventually generates outputs. This view coincides with the integrated perspective of the system (see Section 2.2.2).

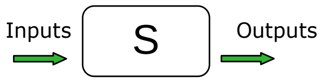


Figure 25 Service specification (SS)

The *service design refined model (SDRM)* is a refinement of the SS behaviour model into a structured behaviour. At this level, we consider the system from its distributed perspective (see Section 2.2.2) as a set of interacting components, for example, components C_1, C_2 and C_3 in *Figure 26*. 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 message exchanges. *Figure 26* shows a simple example in which the input to the system corresponds to an input message I_1 to the component C_1. In this example, after receiving this message, component C_1 generates an intermediate output message O_1, which is then taken as input I_2 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.

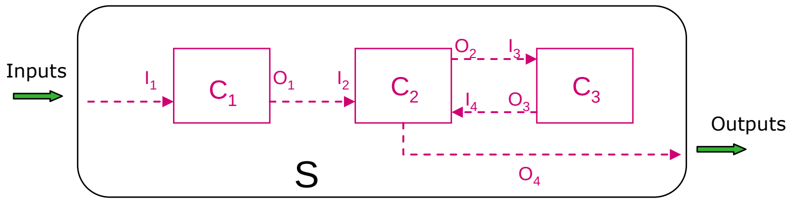


Figure 26 Service Design Refined Model (SDRM)

The *service design component model (SDCM)* is a refinement of the SDRM into a detailed behaviour of concrete components. At this level, we consider again the system from its distributed perspective as a set of interacting components with individual internal processes and activities. *Figure 27* shows that each component has an internal flow of activities in order to provide inputs and outputs for the message exchange.

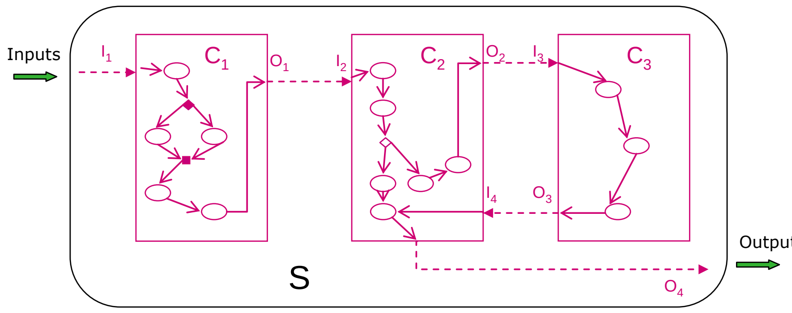


Figure 27 Service Design Component Model (SDCM)

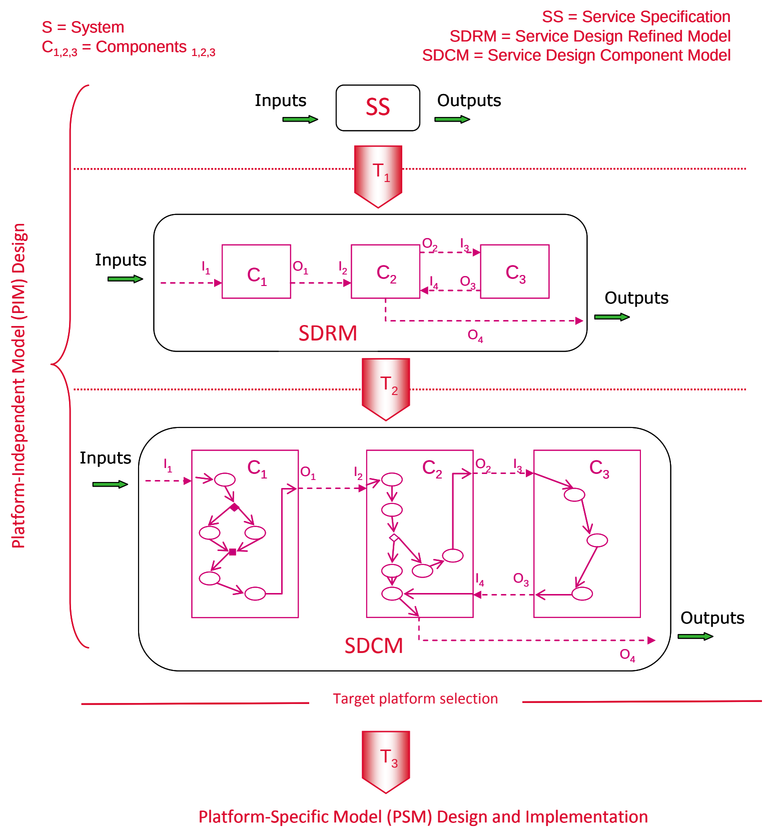
A *platform-specific model design (PSM)* describes the realisation of the application in terms of a specific target technology. Several alternative PSMs may implement a PIM as long as correctness and consistency are guaranteed. Therefore, it is in principle possible to use different middleware technologies to realise the platform-specific service design. Several intermediate steps could be applied also at the PSM level before generating the implementation code. However, since this work focuses on the behaviour modelling of the application at the PIM level, we assume a direct transformation to code and do not discuss the PSM level further.

3.1.2 Model transformations

Our model-driven methodology includes two model transformations at the PIM level, i.e., transformations T_1 and T_2, and one transformation T_3 from PIM to PSM. *Figure 28* relates these transformations to the models described above.

Transformation T1 refines the SS behaviour model, which is too abstract to be directly executed by any platform-specific technology, into a SDRM structured behaviour based on the components of the reference architecture chosen for the application. Although this SDRM model reflects the internal structure of the system and the interactions among components, it is not executable yet. Therefore, transformation T_2 synthesises this structured behaviour into the SDCM behaviour of individual components, which can in principle be executed, since it prescribes the internal activities of each component and how these components interact with each other in order to achieve the goals of the application. Finally, the transformation T_3 maps the SDCM, which is platform-independent, onto some specific middleware platform on which the design can be realised. In principle, it is possible to use different middleware platforms to implement the SDCM.

Figure 28 Model transformations


S = System
$C_{1,2,3}$ = Components $_{1,2,3}$
SS = Service Specification
SDRM = Service Design Refined Model
SDCM = Service Design Component Model
Inputs
SS
Outputs
T_1
SDRM
T_2
SDCM
Target platform selection
T_3
Platform-Independent Model (PIM) Design
Platform-Specific Model (PSM) Design and Implementation

A common requirement to all the transformations in *Figure 28* is that they should preserve correctness and consistency with the original abstract specification of the system. In other words, it is possible to gradually add details to these models to specify the internal view of the system. However, consecutive models should always preserve the original behaviour from the perspective of the external environment (users). This is represented in *Figure 28* where inputs and outputs of a higher level are preserved at a lower level: the level of details gradually increases from SS to SDCM, but the inputs and outputs to/from the system should always be the same. Moreover, if the SS level shows multiple inputs and outputs with a specific ordering/interleaving, this should be preserved as well at lower levels.

Towards the automation of the transformations in *Figure 28*, we used the following approach, consisting of three phases:

1. *manual creation of models*: we initially created our models manually in order to have a clear understanding of the source and target models of our transformations,
2. *manual mappings between models*: we created manual mappings from source to target model in order to generate systematic guidelines for these transformations,
3. *automation of transformations*: we used these guidelines to generate prototypes of transformation specifications that could be taken as input by some transformation engine.

Section 3.2 further elaborates on this approach.

3.1.3 Modelling language

Languages are essential in a methodology and should be chosen properly. Our methodology should prescribe: (1) a language to model the application structure and behaviour at the PIM level, and (2) a language that can be used at the PSM level to implement this structure and behaviour according to the target platform of choice. The choice of a suitable modelling language is not a trivial task, as demonstrated by the discussion in the MDA community on the lack of a commonly agreed language to represent behaviour [11]. Chapter 4 discusses the state-of-the-art in behaviour modelling languages and techniques, compares their strengths and weaknesses, and selects the language(s) that we considered more suitable for the purpose of this thesis.

3.2 Modelling and Transformation Approach

The main challenge in this thesis consists of automating as much as possible the PIM model transformations proposed in our methodology. In order to achieve this, we have looked at ways not only to realise automatic model transformations, but also to reuse the knowledge acquired in the process of automating these transformations. We started by considering a case study on the realisation of a context-aware mobile application, and a modelling language suitable to represent the reactive behaviour of this application. As mentioned in Section 3.1.2, we distinguished three phases of this case study. During phase 1 we manually created the SS, SDRM and SDCM in order to have a clear understanding of the source and target models of our transformations (*Figure 29*).

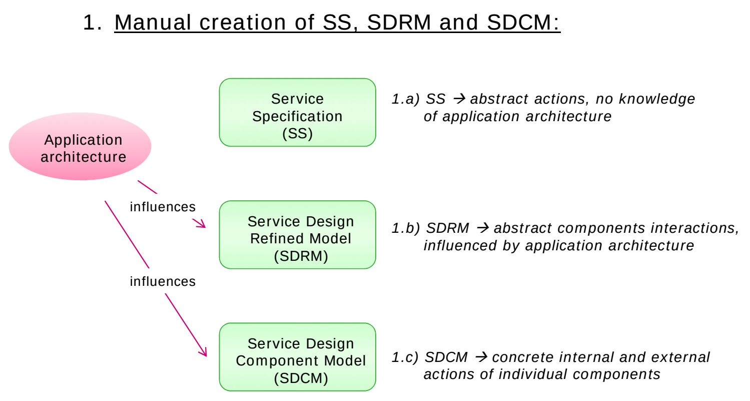

1. Manual creation of SS, SDRM and SDCM:
Application architecture
influences
influences
Service Specification (SS)
Service Design Refined Model (SDRM)
Service Design Component Model (SDCM)
1.a) SS → abstract actions, no knowledge of application architecture
1.b) SDRM → abstract components interactions, influenced by application architecture
1.c) SDCM → concrete internal and external actions of individual components

Figure 29 Modelling and transformation approach: phase 1

Afterwards, during phase 2, we realised a manual mapping of the SS into an SDRM and we noticed some recurrent behaviour in the application that could be generalised and exploited for reuse (step 2.a in *Figure 30*). Particularly, we have been able to identify in the SDRM an entire set of recurrent behaviour execution traces among the components of the system. We called the identified traces *interaction patterns*. Section 3.3 elaborates on the interaction pattern concept.

At this point, we classified all the interaction patterns we were able to identify in the SDRM (step 2.b in *Figure 30*). This classification was based on the type of interaction performed by the pattern and the involved components. We called this classification *interaction markers library* and used it as bottom-up knowledge to mark the abstract actions at the SS level (step 2.c in *Figure 30*). In this way, we created a vertical correspondence of interaction markers in the SS onto interaction patterns in the SDRM (step 2.d in *Figure 30*).

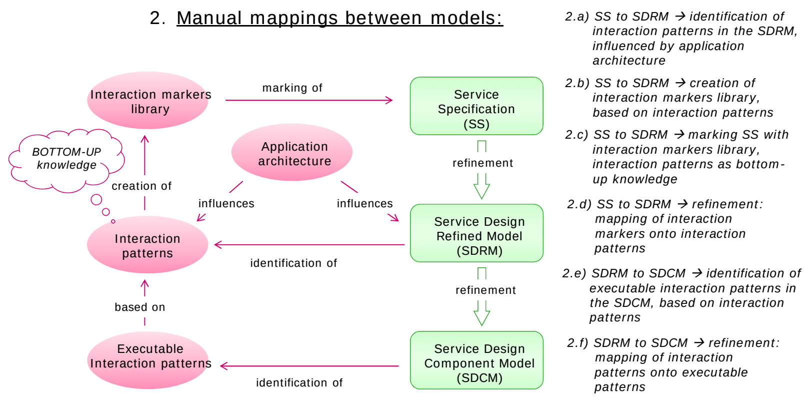
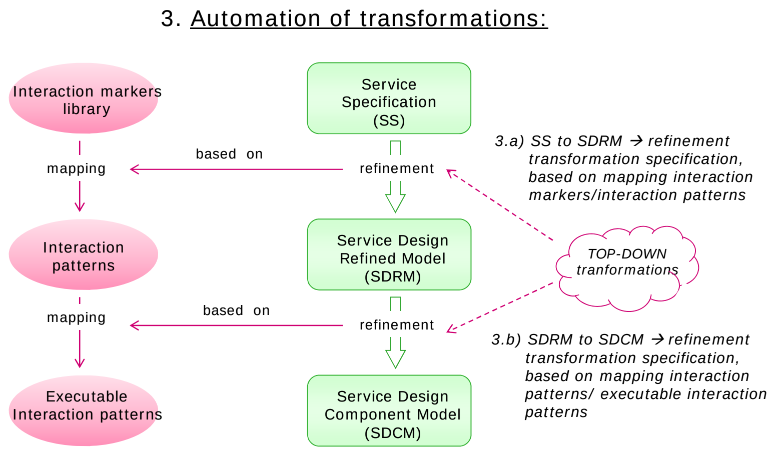

2. Manual mappings between models:
Interaction markers library
marking of
Service Specification (SS)
BOTTOM-UP knowledge
Application architecture
creation of
refinement
influences
influences
Interaction patterns
identification of
Service Design Refined Model (SDRM)
based on
refinement
Executable Interaction patterns
identification of
Service Design Component Model (SDCM)
2.a) SS to SDRM → identification of interaction patterns in the SDRM, influenced by application architecture
2.b) SS to SDRM → creation of interaction markers library, based on interaction patterns
2.c) SS to SDRM → marking SS with interaction markers library, interaction patterns as bottom-up knowledge
2.d) SS to SDRM → refinement: mapping of interaction markers onto interaction patterns
2.e) SDRM to SDCM → identification of executable interaction patterns in the SDCM, based on interaction patterns
2.f) SDRM to SDCM → refinement: mapping of interaction patterns onto executable patterns

Figure 30 Modelling and transformation approach: phase 2

Afterwards, we realised the manual mapping of the SDRM into the SDCM (step 2.e in *Figure 30*) in which we related the interaction patterns identified at the SDRM level to corresponding patterns at the SDCM level that can in principle be executed. This assignment was quite straightforward, since the interaction patterns 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, when scheduling the patterns execution, we could decide to interleave these patterns, by executing all the patterns one at a time in a single thread of control. Alternatively, we could decide to execute these patterns in parallel threads of control. Independently of the choise made, some formalism had to be used to represent and analyse these choices. Therefore, we looked at formalisms to synthesize components behaviour. Chapter 4 discusses these formalisms.

During phase 3, we finally automated our transformations by using the top-down mappings SS to SDRM to SDCM created in the previous step (*Figure 31*).

Figure 31 Modelling and transformation approach: phase 3


3. Automation of transformations:
Interaction markers library
mapping
based on
Service Specification (SS)
refinement
3.a) SS to SDRM → refinement transformation specification, based on mapping interaction markers/interaction patterns
Interaction patterns
Service Design Refined Model (SDRM)
TOP-DOWN tranformations
mapping
based on
refinement
3.b) SDRM to SDCM → refinement transformation specification, based on mapping interaction patterns/ executable interaction patterns
Executable Interaction patterns
Service Design Component Model (SDCM)

3.3 Interaction Patterns

Interaction patterns can be of two different types, namely basic and composite patterns. Basic patterns involve interactions between only two participants, and composite patterns involve interactions between more than two participants. Composite patterns can be obtained by combining basic patterns with the use of logical operators. Therefore, basic interaction patterns can be defined as follows:

Definition 9 Basic interaction patterns

Basic interaction patterns are building blocks of behaviour, defined from the internal distributed perspective of the system, consisting of actions between two interacting components (participants) and the information eventually exchanged between these components.

Figure 32 shows examples of basic interaction patterns identified in the SDRM in *Figure 26:*

- the basic pattern user→component represents a one-way interaction between the participants user and the component C_1 in which the user provides the input message I_1 to C_1,
- the basic pattern component→component represents a one-way interaction between the two participants component C_1 and C_2,
- the basic pattern component←→component represents a two-way interaction pattern between the two participants C_2 and C_3 in which component C_3 sends back a message (input $I_4 \equiv$ output O_3) to C_1,
- the basic pattern component→user represents a one-way interaction between the two participants component C_2 and user.

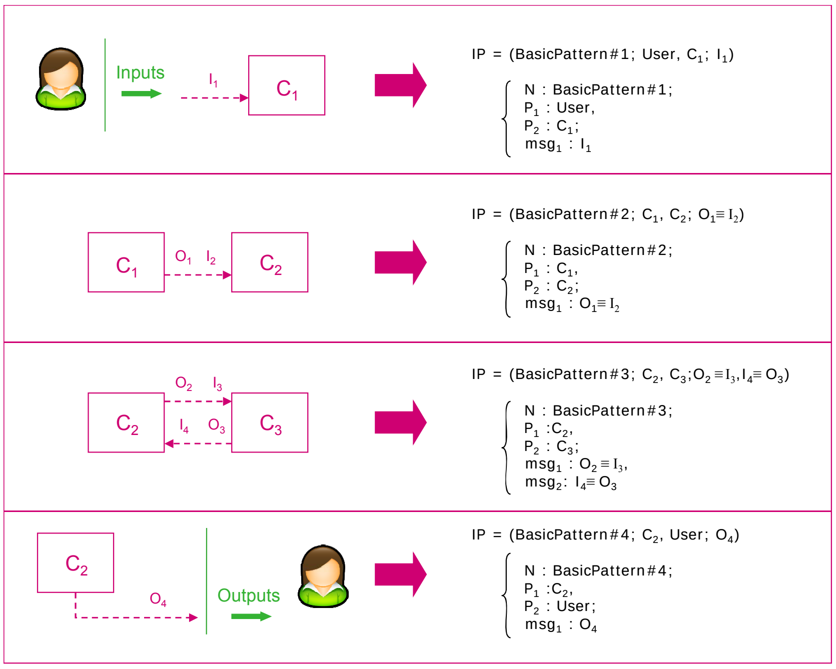


Figure 32 Basic interaction patterns

Definition 10 Composite interaction patterns

Composite interaction patterns are pieces of behaviour, defined from the internal distributed perspective of the system, consisting of a set of building blocks (basic interaction patterns) assembled by using logical connectors in order to achieve a specific goal.

Figure 33 shows examples of how to combine the basic patterns mentioned above in more complex configurations of composite patterns. The first two examples represent concatenations of basic patterns to be executed in sequence. The third example represents an exclusive choice.

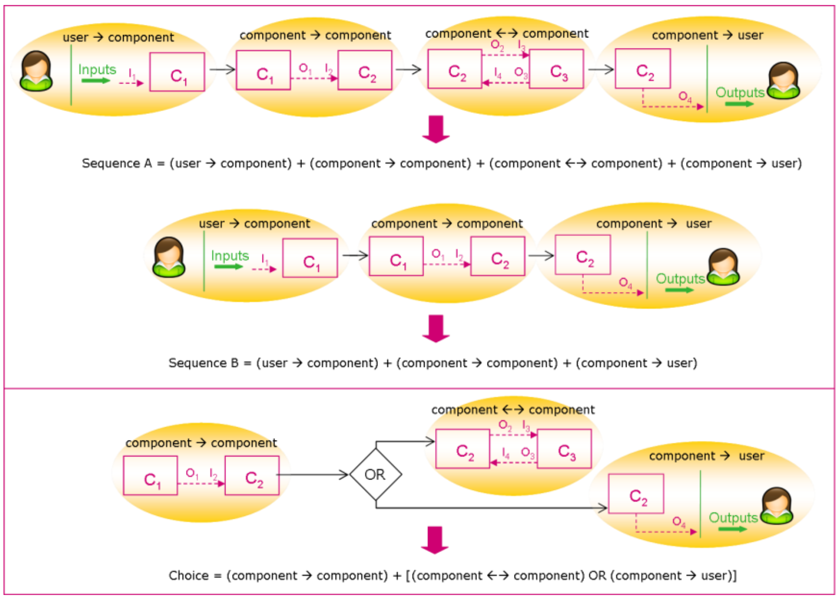

user → component
Inputs
component → component
component ←→ component
component → user
Outputs
Sequence A = (user → component) + (component → component) + (component ←→ component) + (component → user)
Sequence B = (user → component) + (component → component) + (component → user)
OR
Choice = (component → component) + [(component ←→ component) OR (component → user)]

Figure 33 Composite interaction patterns

3.4 Application Architecture

In this thesis we target context-aware mobile applications for two reasons, which take into account a user and a developer perspective, respectively:

1. Users demand advanced and personalised services. Context-aware mobile applications provide this kind of services, which create added-value according to their users' personal preferences and needs, wherever these users are and whatever they are doing.
2. Context-aware mobile applications provide the developers with a representative example of reactive behaviours. These applications are able to monitor the user's context and, in case of changes in this context, consistently adapt their behaviour in order to satisfy the user's current needs or anticipate the user's intentions. Since this work aims at developing a methodology to model the behaviour of systems, context-aware mobile applications are suitable for our purposes. Moreover, because of their inherent complexity and relative immatureness, context-aware mobile applications are in particular need of receiving methodological support for their design.

In order to develop context-aware mobile applications, a middleware platform is necessary that consists of generic components and supports general purpose functions used by such applications. For example, all context-aware mobile applications are capable to retrieve context information from the user's context and, based on this information, provide relevant services to their user. Because of stringent time-to-market requirements, it is not desirable that each individual application captures and processes context information for its own use, since application development in this case would be time consuming and costly. In contrast, with a proper a middleware platform, generic functions can be made available and reused in various context-aware mobile applications. Therefore, the development of such applications relies on dynamic middleware platforms, which consist of a variety of components distributed in the environment that interoperate by making use of each other's services. We have defined a middleware platform based on a reference architecture tailored to context-aware mobile applications. This reference architecture is discussed in Section 3.4.2 (*Figure 35*).

3.4.1 Tiers

In order to organise the components of our middleware platform, a tiered architecture has been defined, consisting of the following tiers (*Figure 34*):

1. a *presentation tier*, which is responsible for the interaction with the user. Interaction can take place through fat terminals (such as notebooks and desktop PCs), thin terminals (such as PDAs), mobile terminals (smart phones and regular mobile phones), and plain-old telephones;
2. an *application logic tier*, which is responsible for enforcing the behaviour of the application. This includes supporting the interaction between different users and coordinating the access to reusable (context, action and information) services;
3. a *resources tier*, which consists of context, action and information services. These services can be offered by external sources (e.g., web services) or can be provided in the scope of user's terminal (e.g., GPS devices acting as context sources).

The coordination task of the application logic tier is assigned to the coordinator, which receives events and triggers actions as reactions to these events. Events may be either *user input events*, which consist of explicit user requests to the application, or *context events*, which consist of relevant changes in the user context. For example, a user input event may be a request for the user's list of buddies, and a context event may be the proximity event triggered whenever a buddy is nearby the user. Actions represent application reactions to user input and context events, and may be an invocation of any internal or external service, such as the generation of a signal, the delivery of a notification or a web service request.

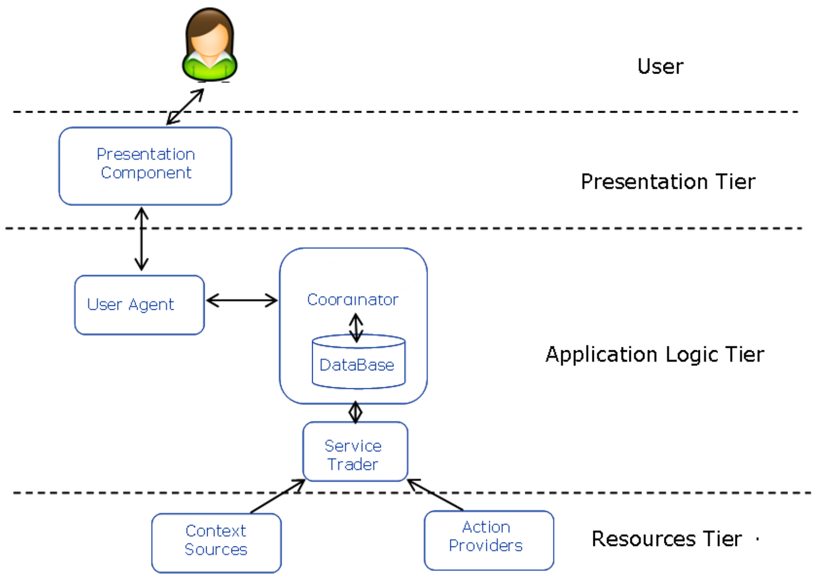


Figure 34 Tiered architecture for context-aware mobile applications

3.4.2 Components

Figure 35 shows our reference architecture, which was originally defined [60-61] for context-aware applications that allow users to contact the right person, at the right time, at the right place, via the right communication channel. However, we assume that this architecture is general enough to be reused with other context-aware mobile applications by simply redefining some application-specific components, such as context sources and action providers. For example, in the health care domain, context sources can be wearable devices that can monitor the user's vital signs, such as heart beat or blood pressure. The use of this architecture does not limit our methodology to context-aware mobile applications, since the methodology can be applied (with minor adjustments) to other categories of applications based on different reference architectures.

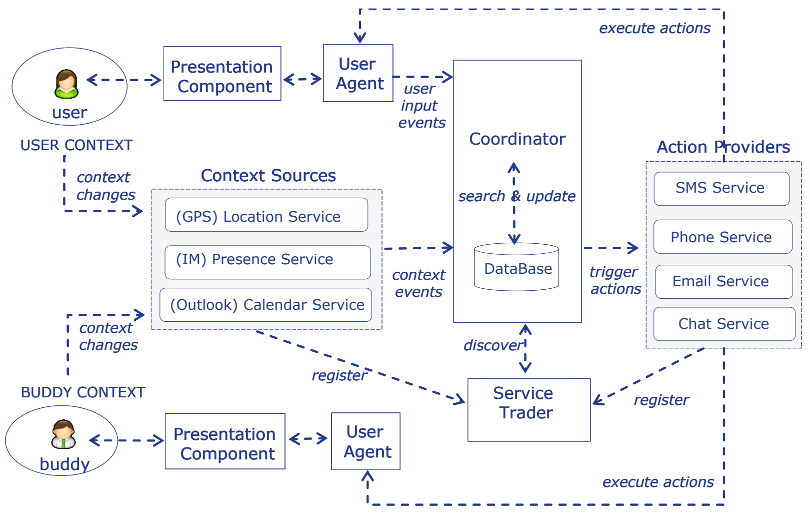


Figure 35 Reference architecture for context-aware mobile applications

Figure 35 shows a single *user* instance interacting with the system and a *buddy* of this user. The *presentation component* takes care of the interactions with the end-user, either the user or her buddy. There is one presentation component for each end-user. Since context-aware mobile applications should be able to provide relevant services to their users anywhere and anytime, we assume that the presentation component is integrated in the user device, which may be either a desktop PC, in the case of users with fixed location, or a Smartphone or Pocket PC phone in the case of users on the move.

The *user agent* in *Figure 35* interacts on behalf of the user with the presentation component to obtain user input and present user output. The user agent is located in the user device and we assume that there is one user agent for each end-user. The user agent also provides the coordinator with user input events.

The *coordinator* in *Figure 35* orchestrates all the other components, with searching and updating a database, which contains information about users (e.g., name, password, preferred contact means and list of buddies). We assume a system configuration with one service coordinator and one database. The service coordinator also interacts with context sources and action providers.

The context sources in *Figure 35* sense changes in the user's context and generates context events for the coordinator. *Figure 35* shows a (GPS) location service that provides information about users' current location, a (IM) presence service that provides indications whether users registered in the application are available online in the network, and a (Outlook) calendar service that provides information about users' appointments and activities. We assume that there is one (GPS) location service, one (IM) presence service and one (Outlook) calendar service for each user agent in this particular configuration. These services are registered in the service trader.

The *action providers* are responsible for performing actions that follow user input and context events. *Figure 35* shows an *SMS service*, *phone service*, *e-mail service* and *chat service*, which enable users to communicate with each other through, respectively, sending messages, making a phone call, sending e-mails and chatting. We assume that there is one SMS service, phone service, e-mail service and chat service for each user agent. These services are also registered in the service trader.

The *service trader* in *Figure 35* registers all the available services offered by context sources and action providers. This allows the coordinator to dynamically discover available services based on the service descriptions that are published in the service trader. After discovering the proper service, the coordinator can invoke it by using the endpoint address contained in the service description. Alternatively, the coordinator can forward this endpoint to the user agent, which can directly invoke the service without intervention of the coordinator. The use of a service trader is a well established pattern of service discovery in service-oriented architectures. Examples of service traders in middleware platforms are the OMG CORBA trader [62] and the UDDI registry [63].

The interactions among components in *Figure 35* are based on the SOA approach, since components are considered only from the point of view of the service that they provide or use without any mentioning of the internal details of how the service itself is implemented. According to service-orientation principles, components make use of each other's services to cooperate in order to support the goals of the application. Therefore, the coordinator in *Figure 35* uses the service offered by context sources, which provide the coordinator with context events. However, the coordinator is not aware of the details of how context sources obtain context information from the user environment and how they process this information in order to generate context events.

3.5 Methodology Overview

Figure 36 shows an overview of the complete methodology, which includes also the roles involved in the development process. These roles are the following:

– the *user* should give support mainly in the initial phase of the methodology, when the application requirements are gathered to specify the expected behaviour of the application. In later phases, the user should also provide feedback to determine whether the executed behaviour, either at the PIM level or PSM level, fulfils the expected behaviour according to the requirements.

– the *designer* is responsible for the behaviour modelling of the application, especially at the PIM level of the methodology. In the initial phase, the designer creates the service specification by using the interaction markers library according to the user requirements.
– the *developer* is responsible for the PSM design of the application and the generation of PSM code.

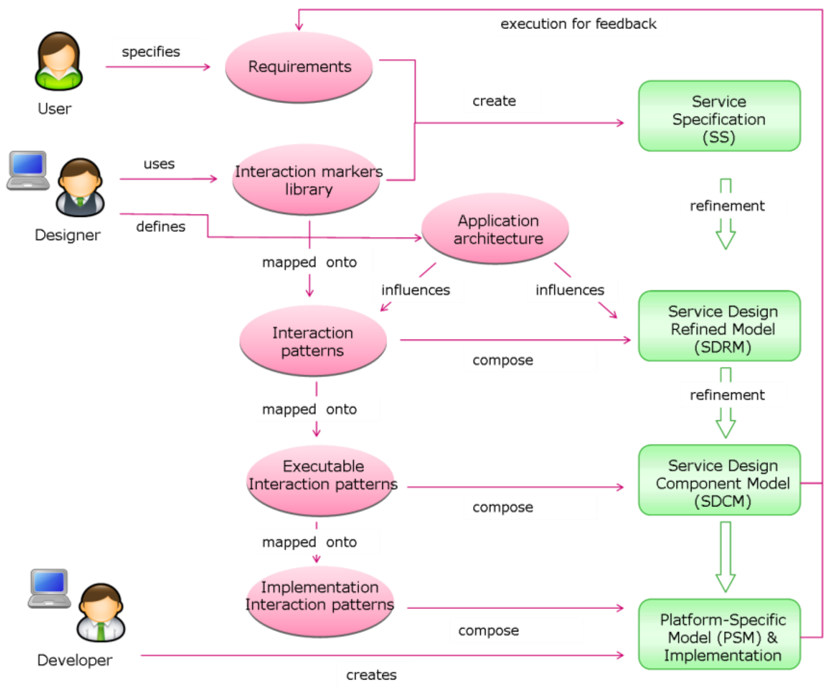

execution for feedback
specifies
Requirements
User
create
Service Specification (SS)
uses
Interaction markers library
Designer
defines
Application architecture
refinement
mapped onto
influences
influences
Interaction patterns
compose
Service Design Refined Model (SDRM)
refinement
mapped onto
Executable Interaction patterns
compose
Service Design Component Model (SDCM)
mapped onto
Implementation Interaction patterns
compose
Developer
Platform-Specific Model (PSM) & Implementation
creates

Figure 36 Methodology: roles, activities, models and dependencies

As depicted in *Figure 36*, the user supports the designer in the creation of the service specification by specifying the application requirements. The designer is provided with the library of interaction markers that we developed in the manual phase of our approach. These markers represent recurrent services that are commonly offered to application users. For example, the marker *simple user request* may represent a service that allows the user to ask the application to perform a certain task for which no application response is required. A user may also require a certain task followed by an application reaction (*user request with response* marker) or by a confirmation of whether the requested task was successfully performed or not (*user request with acceptance or rejection response* marker). Moreover, we have added to the markers library some specific functionality common to all context-aware mobile applications, such as retrieval of context information (*context query* marker), and application reactions to context changes without explicit user intervention (*context event with alert* marker).

By using our interaction markers library, the designer can in principle assemble the behaviour of a new application at a high level of abstraction as a combination of building blocks that already have a complete top-down mapping to the implementation, instead of designing and implementing this application from scratch. In this way, the designer can include an interaction marker from the library in the specification, such as, for example, a *user request with acceptance or rejection response* marker, without any knowledge on how the application works internally to provide the expected response to the user.

Ideally, we aim at allowing the user to assemble the behaviour of a personalised application by combining existing building blocks (SS), automatically obtaining more refined interaction patterns (SDRM), and automatically generating executable interaction patterns (SDCM) that are consistent and correct with respect to the original application behaviour specified in the SS. Realistically, we expect that full automation is in general not achievable with real-life applications. Therefore, we aim at automating as much as possible the transformation steps from SS to SDRM to SDCM mentioned above.

Chapter 4

Behaviour Modelling Techniques

This chapter presents a survey of techniques for behaviour modelling that can be beneficially used in model-driven development and are relevant for this thesis. These techniques use several modelling notations, such as, for example, Transition Systems, Live Sequence Charts, UML, Petri Nets, and BPMN. For each of these techniques, we give an overview and position it with respect to the abstraction levels of our layered methodology. In Chapter 5 we evaluate these techniques based on some qualitative criteria and show how they can be integrated in our methodology.

This chapter is organised as follows: Section 4.2 illustrates a technique for behaviour synthesis using Transition Systems (TSs), Section 4.1 presents a technique based on Live Sequence Charts (LSCs), Section 4.3 discusses a technique for Java code generation from UML-like diagrams, Sections 4.4 and 4.5 present techniques for transforming BPMN process models into Petri Nets, Section 4.6 illustrates a technique based on patterns to transform BPMN process models into BPEL, Section 4.7 discusses a technique for generating BPEL semantics in terms of open Workflow Nets (oWFN), which are a special case of Petri Nets, and, finally, Section 4.8 proposes a technique for BPEL process generation from abstract specifications represented in the Interaction System Design Language (ISDL).

4.1 Synthesis from Properties and Scenarios

The work presented in [64] proposes a technique for behaviour synthesis based on state machines. State machines are models consisting of states, transitions between these states, and actions. These models are suitable to represent behavioural aspects of applications, particularly for the specification of behaviours that are assigned to concrete components, like in the service design component models of this thesis.

In [64], two types of behaviour model synthesis techniques are analysed: synthesis from properties and from scenarios. A behaviour model synthesized from properties provides an upper bound of the modelled application, since it includes all possible acceptable behaviours of the application that do not violate those properties. However, it may not be necessary and advisable to model such a large set of behaviours (all these behaviours are possible, but not all of them are required). In contrast, a behaviour model synthesized from scenarios provides a lower bound of the modelled application, since it includes a limited set of example behaviours that the application can assume. However, this set may considerably grow when extending the scenario (these example behaviours are required, but there are other possible behaviours that have not been considered yet). Therefore, [64] suggests that a comprehensive behaviour model should be synthesized both from properties and scenarios.

As explained in [64], traditional state machines models, such as Labelled Transition Systems (LTSs), cannot capture this middle ground between properties and scenarios, since LTSs do not support the distinction between required and possible behaviours. Therefore, a formalism based on Modal Transition Systems (MTSs) is proposed in [64] that allows to distinguish possible from required behaviour, preserving the original properties and scenario, and also supporting elicitation of new properties and scenarios. More details on LTSs and MTSs can be found in [64]. *Figure 37* shows an overview of the behaviour synthesis technique from properties and scenario.

The left part of *Figure 37* shows the synthesis from properties. In [64], safety properties are considered, i.e., those properties that specify that "nothing bad can happen". Although several formalisms can be in principle used to express these properties, Fluent Temporal Logic (FLTL) is recommended in [64-65] because FLTL provides a uniform framework for specifying and model checking state-based temporal properties and event-based models. Using the algorithm suggested in [64], an LTS can be generated from a FLTL property. This LTS can be further synthesized in an MTS, which distinguishes *possible* from *required* behaviours.

The right part of *Figure 37* shows the synthesis from scenarios. Although several notations can be in principle used to represent scenarios, Message Sequence Charts (MSC) [66] are proposed. Based on the technique presented in [67], a scenario is represented using a combination of basic message sequence charts (bMSC) and high-level message sequence charts (hMSC). A bMSC represents a scenario in an UML sequence diagram-like notation, i.e., as sequences of interactions among components. An hMSC represents a scenario in an UML activity diagram-like notation, i.e., as a flow of activities, each of them described as a bMSC. Using the algorithm in [67], it is possible to synthesize an LTS from an MSC scenario. Afterwards, using the algorithm described in [64], it is further possible to synthesize an MTS, which distinguishes *possible* from *required* behaviours.

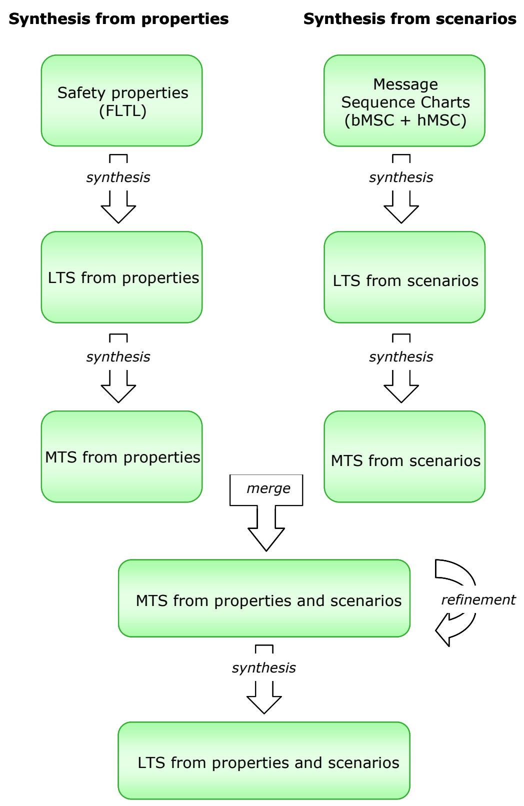


Figure 37 Overview of the technique for behaviour synthesis from properties and scenarios

The two MTSs synthesized from properties and scenarios can be merged in one MTS, as shown in *Figure 37*. The merged MTS preserves the original properties and scenarios, as demonstrated in [64]. In principle, this MTS can be incrementally refined by adding new properties to reduce the *possible* transitions to *required* transitions. Possibly, a final LTS with only *required* transitions is generated. In practice, it may not be necessary to do so since the designer may explicitly decide to leave some behavioural choices (possible transitions) open further down in the development process.

The MTSA (Modal Transitions System Analyser) [68] is a prototype tool to support the elaboration and verification of MTSs. The MTSA is built on top of the LTSA (Labelled Transitions System Analyser) tool [69], which allows to automatically model check required properties in an LTS and supports simulation of the system behaviour. Based on the work in [70], some LTSA plug-ins further allow to validate the executable behaviour of web services compositions represented in BPEL code and verify whether this BPEL code fulfils the requirements represented in corresponding MSC scenarios.

Relevance to this thesis

Figure 38 positions the synthesis from properties and scenarios technique at the PIM level design of our methodology.

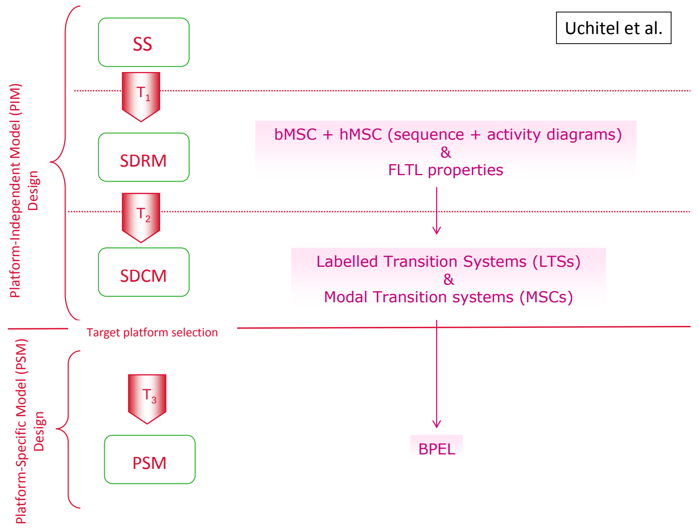

Uchitel et al.
Platform-Independent Model (PIM) Design
SS
T1
SDRM
T2
SDCM
bMSC + hMSC (sequence + activity diagrams) & FLTL properties
Labelled Transition Systems (LTSs) & Modal Transition systems (MSCs)
Target platform selection
Platform-Specific Model (PSM) Design
T3
PSM
BPEL

Figure 38 Positioning of Uchitel et al. with respect to our abstraction levels

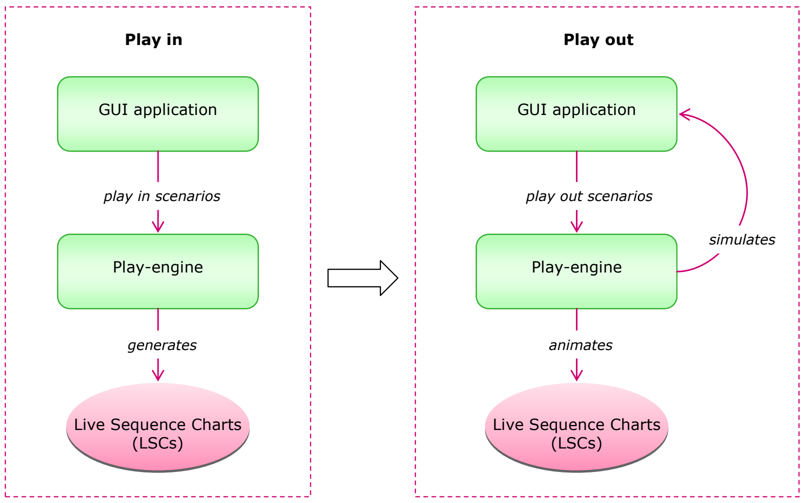
The representation of scenarios in terms of MSC sequence and activities diagrams notation corresponds to our SDRM level, in which the system behaviour is assigned to application components interacting with each other. The synthesis of these components in terms of LTSs and MTSs from the perspective of one specific component corresponds to our SDCM model. In [64] the synthesis of behaviour specifications is discussed, but not how these behaviours can be executed. However, since the work in [70] suggests BPEL as a possible target technology, we positioned BPEL at the PSM level in *Figure 38*.

4.2 Play-in Play-out

The work in [71-72] describes a technique called the *play-in/play-out approach* for specifying (pieces of) behaviours in a user-friendly way at a high abstraction level and executing them at a lower abstraction level. This technique is tailored to reactive systems. Since the context-aware mobile applications considered in this thesis are an example of reactive systems, the play-in/play-out approach is relevant for our work.

As schematically represented in *Figure 39*, the *play in* phase of the approach allows one to specify scenarios in a graphical interface of the system under development and automatically generate a corresponding Live Sequence Chart (LSC) [73]. The transformation is automatically realised by a tool called the *play engine*. By allowing one to specify system requirements in user-friendly way, i.e., clicking buttons and rotating knobs in a GUI, the abstraction level in the requirements specification process is raised. In contrast, specifying these requirements in a formal language would require specific expertise and detailed knowledge of the language's syntax and semantics.

Figure 39 Overview of the play-in/ play-out approach



LSCs are a scenario-based visual formalism that extends the ITU message sequence charts (MSCs) [66] and their UML sequence diagrams variant [22]. As discussed in [72], LSCs have been created since MSCs and their UML variant have an extremely weak partial-order semantics that does not allow representing exhaustively behavioural requirements of a system. In contrast, LSCs have a powerful formal semantics [73], which allows one to distinguish scenarios that *may* happen, called *existential charts*, from scenarios that *must* happen, called *universal charts*. LSCs can also specify messages that *may* and *must* be received, which are called *cold* and *hot* messages, respectively. In addition, also conditions can be *cold*, i.e., they may be true otherwise the control moves out of the considered chart, or *hot*, i.e., they must be true otherwise the system aborts.

As further shown in *Figure 39*, the *play out* phase of the approach allows one to specify scenarios in the graphical interface and test the behaviour of the system. The underlying idea is that one should act as the end-user of the system, i.e., using the system GUI like if it was the final system, which does not require any knowledge about LSCs or the scenarios specified in the play in phase. As a consequence of end-user actions, the play engine animates the LSCs specification and simulates the system reactions on the GUI. In this way, the end-user can evaluate whether the system behaves as expected or not. *Figure 40* shows the benefits of applying this technique in the scope of this thesis.

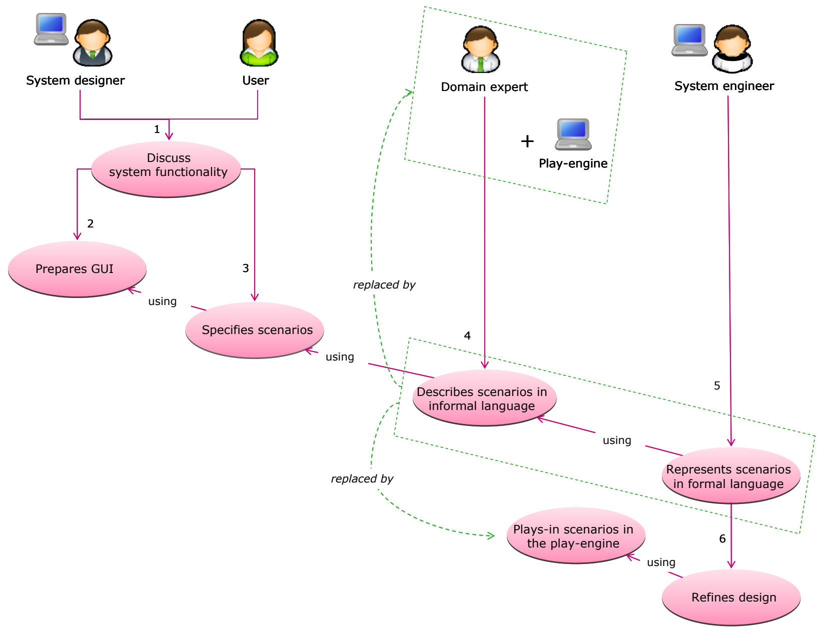


Figure 40 Playing in behaviours

Figure 40 shows that when developing a new system, the system designer and the end-user should discuss the system functionality and prepare a simple GUI, in which the user can specify scenarios [72]. In traditional methodologies these scenarios are taken as input by a domain expert, who writes them in an informal language (phase 4 in *Figure 40*). This informal specification is translated into a formal specification by a system engineer (phase 5 in *Figure 40*), who can further refine it into more technical details towards the implementation. The benefit of the play-in/play-out approach is that the domain expert can specify the scenarios directly in the play engine, which automatically creates the corresponding formal specifications in terms of LSCs. This benefit is represented in *Figure 39* by merging the phases 4 and 5 in a single phase.

Current work [74] within the play-in/play-out approach focuses on the realisation of *PlayGo* [75], which is an extended and broader elaboration of the play engine tool. PlayGo is an Eclipse-based comprehensive tool for scenario-based programming with a compiler that transforms LSCs into AspectJ code [76].

Relevance to this thesis

Figure 41 positions the play-in/play-out approach with respect to our abstraction levels.

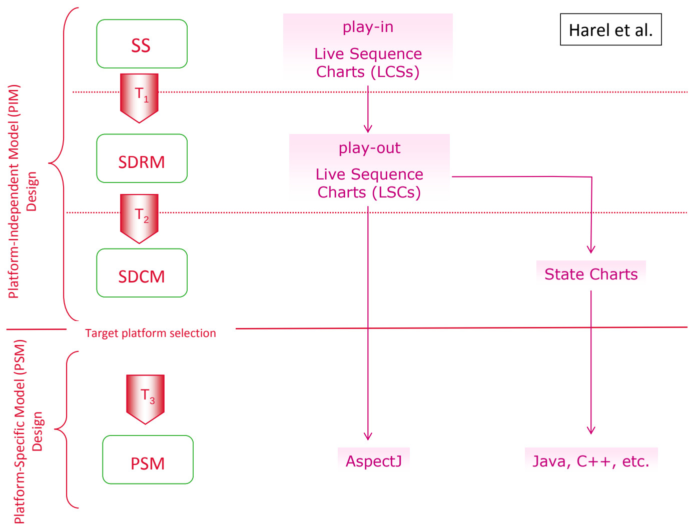

Harel et al.
SS
T1
SDRM
T2
SDCM
PSM
T3
play-in
Live Sequence Charts (LCSs)
play-out
Live Sequence Charts (LSCs)
State Charts
AspectJ
Java, C++, etc.
Target platform selection
Platform-Independent Model (PIM) Design
Platform-Specific Model (PSM) Design

Figure 41 Positioning of Harel et al. with respect to our abstraction levels

Since the play-in phase of the approach allows the designer (or even the final user) of the system to specify the application requirements at a high abstraction level, it can be compared to our SS level. The play-out phase can be then compared to our SDRM level, in which we represent the cooperating behaviour of interacting components without any knowledge of how these components behave internally. In [71], the behaviour of interacting components is also called *inter-object* behaviour, while the internal behaviour of individual components is called *intra-object* behaviour. Our SDCM level represents the intra-object behaviour of our components synthesized from the inter-object behaviour represented in the SDRM level. In contrast, the play-out mechanism executes the inter-object behaviour of the system for simulation purposes, but does not further synthesize intra-object behaviour. Therefore, the play-in/play-out approach only covers our SS and SDRM levels. However, it could be possible to generate SDCM-like models by translating LSCs diagrams into state charts [77], which allow to represent the intra-object behaviour, and then generate code out of these state charts corresponding to our PSM level, by using available tools, such as Rhapsody [78]. The playGo tool currently under development provides a straightforward way to generate AspectJ code from LSCs, i.e., realises a transformation from our SDRM level directly to the PSM level, as indicated in *Figure 41*.

4.3 Story-Driven Modelling

The Fujaba Project [79] aims at supporting model-driven development by providing an environment that automatically generates code from abstract design models. These models should be complete in order to represent both the structural and behavioural aspects of the software application under development. Moreover, the transformation from design models to code should be semantics-preserving in order to assure that the target model of the transformation still conforms to the behavioural requirements specified in the source model. Both these challenges of representing complete behavioural models and creating semantics-preserving transformations are addressed in the Fujaba project. *Figure 42* shows the technique for modelling application structure and behaviour and automatically generate Java source code by using the Fujaba tool.

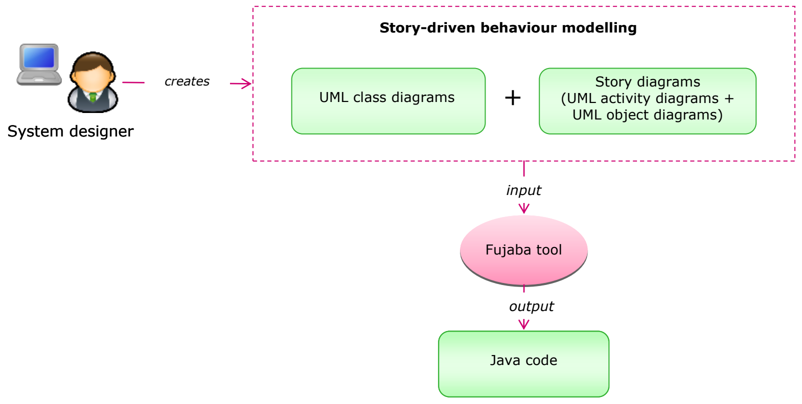

Story-driven behaviour modelling
System designer
creates
UML class diagrams
+
Story diagrams (UML activity diagrams + UML object diagrams)
input
Fujaba tool
output
Java code

Figure 42 Overview of the story-driven modelling technique

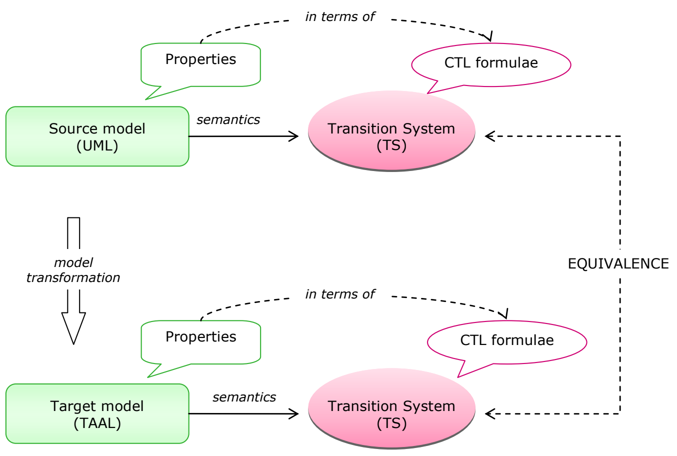
As depicted in *Figure 42*, a system designer (who is also an expert of the problem domain) should model the desired application as a combination of UML class diagrams, which represent the application structure, and *story diagrams* [80], which represent the application behaviour. Story diagrams are a combination of UML activity diagrams, which represent the control flow of methods represented in the class diagrams, and the so called *story patterns* [80], which represent the internal behaviour of these methods. Story patterns can be specified as Java code or graph rewrite rules [81], which are graphical rules with a left-hand side (LHS) and a right-hand side (RHS). In case the LHS of a rule is found in a graph (*pattern matching*), this LHS is replaced by the corresponding RHS. In the Fujaba technique, story patterns are expressed in a UML object diagram-like notation.

Figure 42 shows that the story-driven behaviour modelling phase is followed by the automatic generation of an executable prototype in Java code. This code generation is realised by the Fujaba tool, which takes the

UML structural and behavioural specifications as input, and maps them onto the Java language. Particularly, UML classes in the structural model are straightforwardly mapped onto Java classes, the control flow in the UML activity diagrams is mapped onto Java control structures, and actions/guards in the story patterns (UML collaboration diagrams) are mapped onto corresponding Java code.

In order to guarantee that the transformation depicted in *Figure 42* preserves the application behaviour specified in the modelling phase, i.e., this transformation is correct, it is necessary to guarantee that the target model does not change the semantics of the source model in an unintended way. The work in [82] aims at proving this correctness by defining a formal semantics of the transformation source and target models in terms of Transition Systems (TSs). As shown in *Figure 43*, informal properties in the source models can be formally expressed in terms of, for example, Computation Tree Logic (CTL) formulae, and compared to properties in the target model also expressed as CTL formulae. When properties in the source model also hold in the target model, then the source and target models are equivalent and the model transformation preserves behaviour correctness.

Figure 43 Behaviour correctness preserving approach



The source and target models in [82] are represented, respectively, in UML activity diagrams and TAAL [83], which is a Java-like object oriented programming language. The transformation from UML activities to TAAL is defined as a graph transformation, which can be executed using the Groove tool [84]. Eclipse plugs-in are available to generate transition systems from UML activities and TAAL. These transition systems correspond to the source and target models, and can be used by Groove to determine whether these two models are trace equivalent or not or, analogously, whether the model transformation preserves semantics.

Relevance to this thesis

Figure 44 positions the story-driven modelling technique with respect to our abstraction levels. The UML activity and object diagrams used to model stories can be positioned at the PIM level and correspond to our SDCM model. These diagrams are used to generate Java code at the PSM level, as shown by the vertical arrow in *Figure 44*. In order to check whether this vertical transformation preserves semantics, the considered technique includes two horizontal transformations at the PIM and PSM levels, respectively. These transformations create transitions systems corresponding to UML activity diagrams at the PIM level, and TAAL programs at the PSM level. TAAL is a simplified and limited version of the Java language.

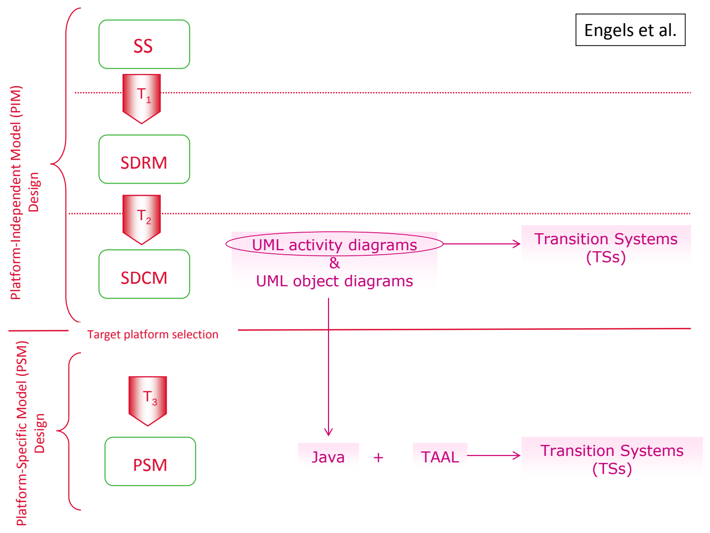


Figure 44 Positioning of Engels et al. with respect to our abstraction levels

4.4 Automated Verification of BPMN Processes

The work presented in [85] proposes a framework, called the *repository*, for automatic verification of business process models represented in several notations, such as, for example, BPMN and BPEL, and software models expressed as UML activity, sequence and state diagrams. Formal verification of these models is important, especially in early stages of the software development, to detect serious design errors before starting the actual implementation of the software. However, most of the existing verification tool cannot be used directly with these models, since these tools are usually based on other type of formalisms, such as, for example, Petri Nets and process algebra. Therefore, the repository framework provides an environment that allows transforming specification models to equivalent models represented in some other formalism so that that these models can be automatically analysed. The repository framework is implemented as a web application through which the users can upload models to a repository and invoke analysis or transformation tools on these models. *Figure 45* schematically shows the technique presented in [86], which transforms BPMN process models into other formalisms for behaviour analisys purposes.

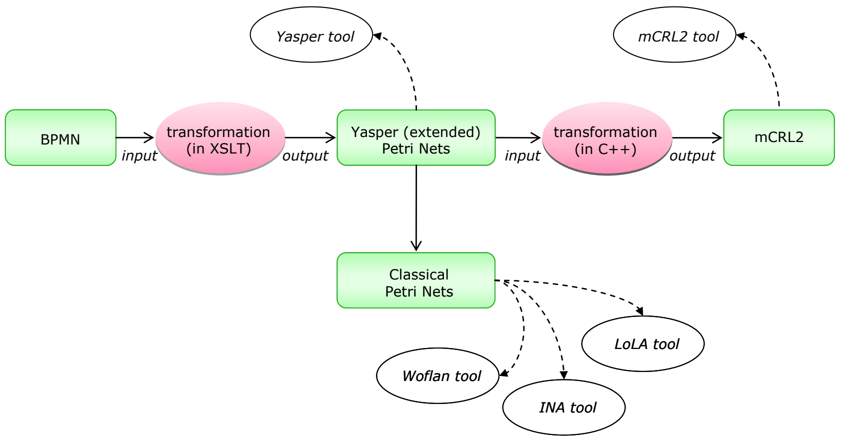


Figure 45 Model transformation technique for behaviour analysis in the *repository* framework

Figure 45 shows that a first transformation step consists of mapping BPMN models onto Yasper Petri Nets [87], which extend classical Petri Nets with some constructs that allow the represention of more complex behaviours, such as inibithor and reset arcs. The mapping rules are presented in [86]. This transformation is implemented in the Extensible Stylesheet Language Transformations (XSLT) [88], which is a declarative XML-based language used for the transformation of XML documents. The Yasper tool [87] allows manual and automatic simulation of Petri Nets, also for performance analysis purposes, verification of soundness properties, and reduction techniques to generate models that are smaller and easier to analyse by preserving soundness and liveness properties. In order to use other Petri Nets verification tools, such as Woflan [89], INA [90] and LoLA [91], these extended Petri Nets should be transformed into classical Petri Nets. That is because these tools cannot be used when inibithor and reset arcs are used in the extended Petri Nets models. Therefore, the solution proposed in [87] and depicted in *Figure 45* consists of transforming these extended Petri Nets into mCRL2 [92], which is a process algebraic formalism. This formalism can be processed using the mCL2 tool [93], which generates a transition system (TS) that can be analysed, for example, in order to find deadlocks and livelocks.

Relevance to this thesis

Figure 46 shows that we considered the transformations supported by the repository framework as horizontal transformations that can be positioned at the SDCM abstraction level of our approach.

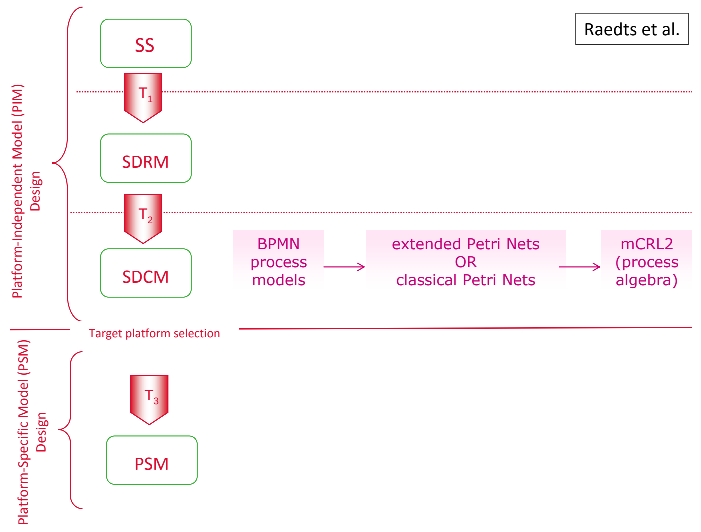

Raedts et al.
SS
T_1
SDRM
T_2
SDCM
BPMN process models
extended Petri Nets OR classical Petri Nets
mCRL2 (process algebra)
Platform-Independent Model (PIM) Design
Target platform selection
Platform-Specific Model (PSM) Design
T_3
PSM

Figure 46 Positioning of Raedts et al. with respect to our abstraction levels

The transformations depicted in *Figure 46* are not refinements, which convert from a model at a certain abstraction level to a more detailed model at a lower abstraction level, but migration transformations, which convert a model in a certain language to an equivalent model in another language at the same abstraction level. By using this type of transformation, the repository framework allows performing analysis that otherwise would not be possible, such as automatic verification and validation of behavioural properties on BPMN models. We have positioned these transformations at the SDCM level, since they are applied to process models that describe the concrete behaviour of processes that participate in business interactions. This corresponds to our component model, in which each component of our architecture performs a process with internal activities.

4.5 A Formal Semantics for BPMN Analysis and Execution

The Business Process Modelling Notation (BPMN) [94] is a standard notation for modelling business processes promoted by OMG. Although the

BPMN syntax is comprehensively documented in a specification document [95], the BPMN semantics is only described informally in this specification. Therefore, the work proposed in [96] aims at defining a formal semantics for BPMN. This semantics is then used both for analysis and execution purposes [97-98]. *Figure 47* shows a transformation from BPMN to Petri Nets, which allows BPMN models to be formally analysed in terms of their semantic correctness, for example to detect deadlocks and livelocks. *Figure 47* shows a transformation from BPMN to a YAWL [99], which is a workflow definition language that extends Petri Nets with several high-level features. This transformation allows BPMN models to be executed in a YAWL workflow engine and be analysed by simulation, animation and execution.

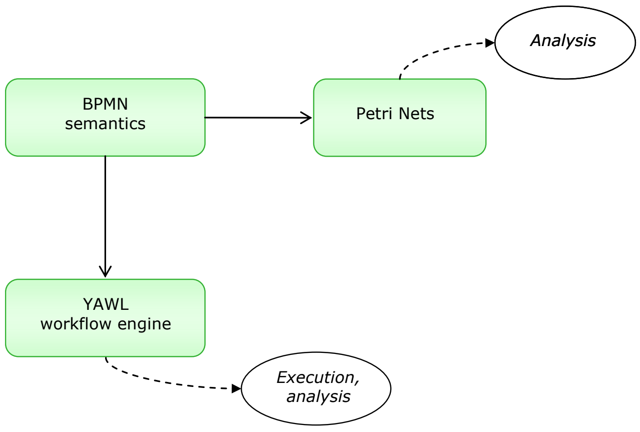


Figure 47 BPMN transformation technique for analysis and execution

In [97], the transformation from BPMN (1.0) to the Petri Net formalism is presented. *Figure 48* shows that this transformation takes as input a BPMN model in XMI [23], which is a standard file format for storing models. In principle, XMI models that conform to the same meta-model are tool-independent and can be seamlessly exchanged. This BPMN model is automatically formalized in terms of a Petri Net according to the mappings in [97], and exported in the Petri Net Markup Language (PNML) notation [100], which is a standard file format to store Petri Nets models. The PNML document can be used as input to Petri Nets analysis tools, such as ProM [101], in order to verify some properties, such as, for example, soundness, which states that for each state that can be reached from the initial state of a process, a firing sequence exists that brings the system to its final state.

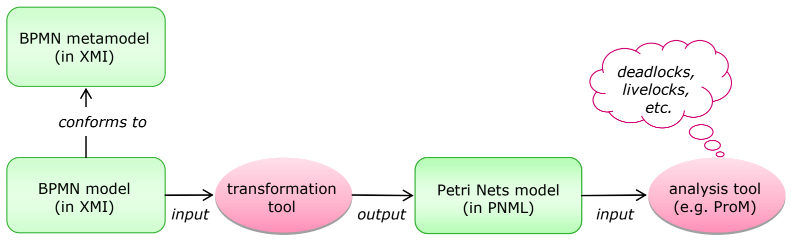


Figure 48 BPMN to Petri Nets transformation for behaviour analysis

In some cases, such as for parallel multi-instance activities and OR-join gateways, no mapping from BPMN to Petri Nets is possible, as explained in [97]. Therefore, it is advisable to transform BPMN to some other formalism, such as, for example, YAWL nets, which can be analysed for correctness and, in addition, can also be simulated, animated and executed. In order to achieve this, the work in [98] defines an execution semantics for a subset of BPMN (2.0) in terms of graph rewrite rules, which are the basic building blocks of graph transformations. As mentioned in Section 4.3, graph rewrite rules consist of a left-hand side, which defines the condition in which a rule should be applied, and a right-hand side, which defines what should be realised when the left-hand side is fulfilled. As depicted in *Figure 49*, the formalization of BPMN in terms of graph rewrite rules can be used to check the conformance of tools that implement the BPMN execution semantics, such as the YAWL workflow engine of [96].

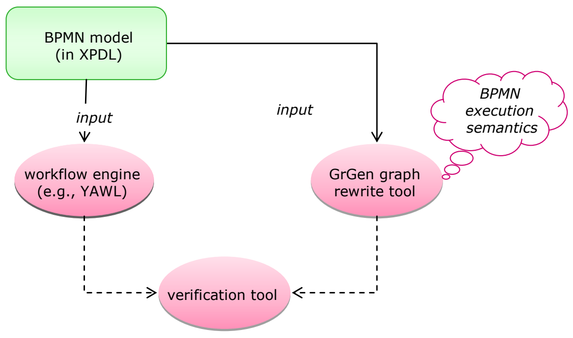


Figure 49 Conformance verification of workflow engines

Figure 49 shows that a BPMN model expressed in the XML Process Definition Language (XPDL) [102] is used as input to a workflow engine and the GrGen tool [103]. The XPDL format is used to facilitate the interchange of business process models between multiple tools during the business process lifecycle. Most BPMN editors allow exporting BPMN models in XMI to the XPDL format. The GrGen tool implements the BPMN execution semantics proposed in [98] in terms of graph rewrite rules. As depicted in *Figure 49*, a verification tool can be used to verify whether the execution behaviour of the model in the engine fulfils the semantics described as graph rewrite rules.

Relevance to this thesis

Figure 50 shows that we considered the transformation from BPMN process models to Petri Nets as a horizontal transformation between models at the same abstraction level, since BPMN models are transformed in semantically equivalent models that can be formally analysed to verify their behaviour correctness. Analogously to the transformation described in Section 4.4, we positioned this horizontal transformation at our SDCM level, which considers an application from the perspective of the internal behaviour of interacting components (processes). *Figure 50* also shows that the transformation from BPMN process models to YAWL can be positioned as a vertical transformation between models at different abstraction levels, since these models can be executed using the YAWL workflow engine.

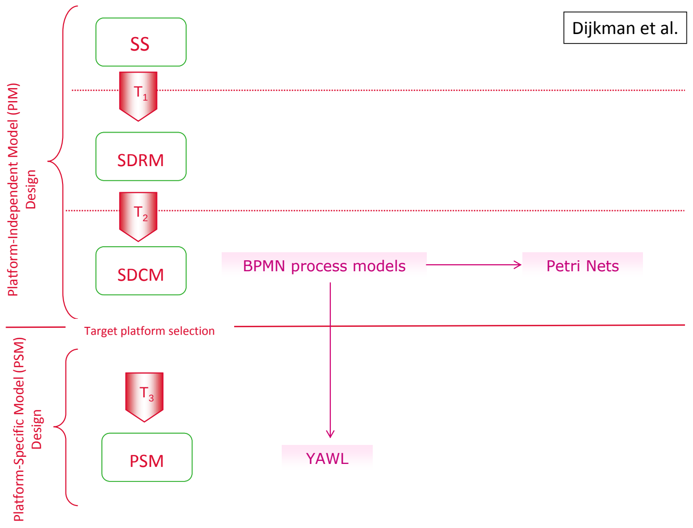

Dijkman et al.
SS
T1
SDRM
T2
SDCM
BPMN process models
Petri Nets
Platform-Independent Model (PIM) Design
Target platform selection
Platform-Specific Model (PSM) Design
T3
PSM
YAWL

Figure 50 Positioning of Dijkman et al. with respect to our abstraction levels

4.6 A Pattern-Based Technique from BPMN to BPEL

The work in [104] provides mappings from BPMN to BPEL [105]. The motivation of this mapping is that, on one hand, BPEL is the *de facto* standard for implementing business processes on top of web services, but it is not appealing for analysts and designers in early stages of the process lifecycle. On the other hand, BPMN is well understood by business analysts and designers, since allows high level representation of business processes. However, BPMN is not executable (yet) by workflow engines. Therefore, the mapping from BPMN to BPEL should bridge the gap between business specialists, who specifies processes at high-abstraction levels in BPMN and technical experts, who implement these processes in BPEL. *Figure 51* shows the technique presented in [104], which has been developed in parallel to the work in [106]. This technique is based on the transformation of graph-oriented structural patterns in BPMN onto block-structured BPEL code.

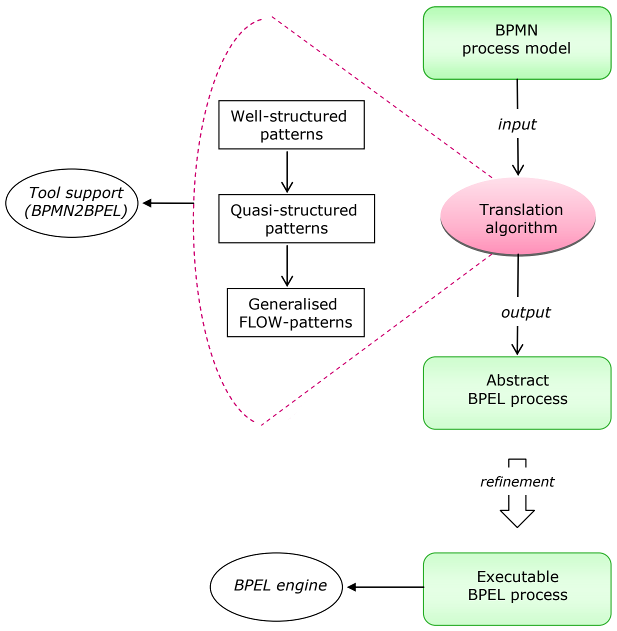


Figure 51 Pattern-based technique from BPMN to BPEL

Figure 51 shows that a BPMN process model is taken as input for the algorithm in [104], which processes the following three types of patterns:

1. *well-structured patterns*, such as "sequence" and "while", which can be directly mapped onto block-structured BPEL constructs,
2. *quasi structured patterns*, which can be easily reduced to well-structured patterns and then translated to block-structured BPEL constructs, and
3. *generalised flow-patterns*, which need to be mapped onto combinations of block-structured BPEL constructs with some additional control links.

The approach used by the algorithm consists of identifying well-structured patterns in the BPMN process model (first the "sequence" well-structured patterns and then all the other well-structured patterns), provide their BPEL translation, and then fold these patterns into an atomic task. In case there are no well-structured patterns left, the algorithm further looks for quasi structured patterns. In case these patterns are all processed, the algorithm finally searches for generalised flow-patterns, if there are any. An Eclipse plug-in called *BPMN2BPEL* realises the automatic transformation of most of the patterns described above.

As shown in *Figure 51*, the output of this mapping is an abstract BPEL process, which captures the interactions with other processes (services), but lacks the internal behaviour that allows this process to be executed by a BPEL engine. Therefore, this abstract BPEL process is a skeleton that should be filled with further details towards an executable implementation.

Relevance to this thesis

Figure 52 shows that the BPMN process models used as source models for the transformation described above can be positioned at the SDCM abstraction level of our methodology. These BPMN models are then translated with a vertical transformation to BPEL in order to be executed. This BPMN to BPEL transformation is equivalent in purpose, i.e., execution using a workflow engine, to the vertical transformation from BPMN to YAWL depicted in *Figure 50*.

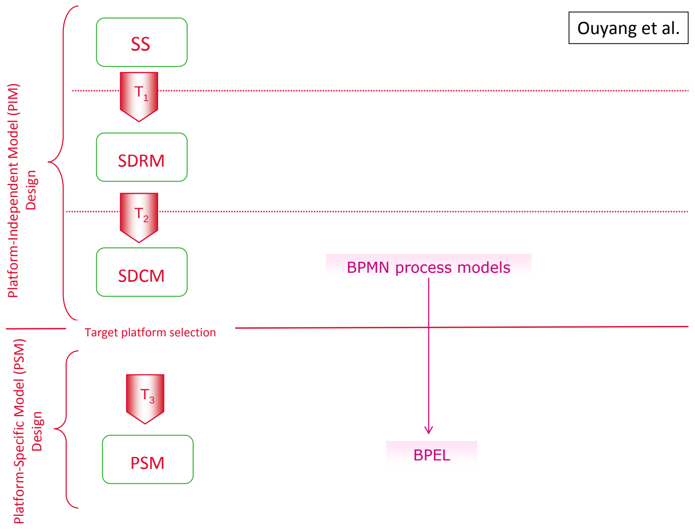

Ouyang et al.
SS
T1
SDRM
T2
SDCM
BPMN process models
Platform-Independent Model (PIM)
Design
Target platform selection
Platform-Specific Model (PSM)
Design
T3
PSM
BPEL

Figure 52 Positioning of Ouyang et al. with respect to our abstraction levels

4.7 Formal Analysis of BPEL Processes using oWFN

A BPEL process can be considered as a workflow enhanced by an interface description that specifies how this process interacts with other processes, called *partners*. In order to guarantee that a process and its partners interact properly, the work in [107] proposes a technique to transform BPEL to open workflow Nets (oWFN) [108], which are a special case of Petri Nets that can be used to model the behaviour of a process interacting with other processes. *Figure 53* shows that this technique transforms a BPEL process to oWFN model in order to verify that the process interacts correctly with its partners, and to classical Petri Nets in order to analyse the internal behaviour of the process.

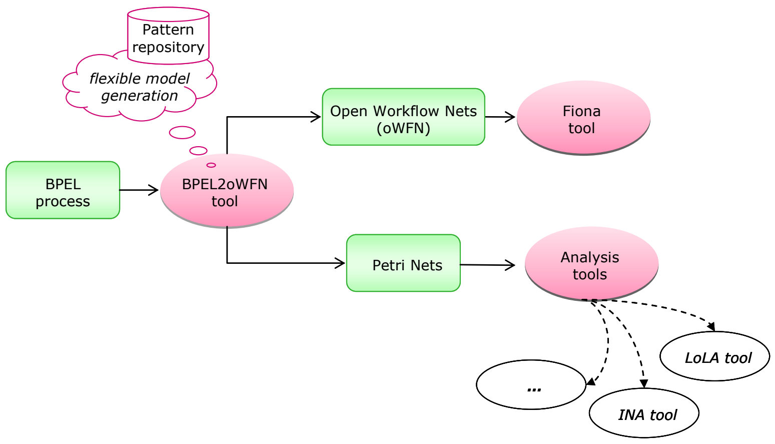


Figure 53 Transformation technique for formal analysis of BPEL processes

The BPEL2oWFN tool [109] in *Figure 53* transforms a BPEL process in an oWFN model applying the so called approach of *flexible model generation* to reduce the size of the generated oWFN model. With respect to a specific property to be analysed, this approach minimizes the model as follows:

1. during the BPEL to oWFN transformation, the tool parses the BPEL code in order to match it against some oWFN patterns stored in a pattern repository, and
2. after the transformation, the tool applies structural reduction rules to the generated model.

A collection of oWFN patterns gives the semantics for BPEL. This semantics is complete, since it covers standard BPEL behaviour, exceptional behaviour, such as fault and compensation, and all data aspects. In this way, [107] provides a formal semantics for BPEL in terms of oWFN and a means to create compact models that can be easily used for computer-aided verification. *Figure 53* shows that the generated oWFN model can be analysed by the Fiona tool [110]. *Figure 53* also shows that the BPEL2oWFN tool supports the transformation to Petri Nets in several file formats, such as LoLA, INA, PNML, etc., which can be analysed by common model checking tools.

Relevance to this thesis

In *Figure 54* we position the formal analysis of BPEL using oWFN at the PSM level of our methodology, since we consider BPEL as a specific technology for implementing business processes and not a modelling language. The purpose of the work in [107] is to map the interactional behaviour of BPEL processes onto oWFN, and the internal behaviour of these processes onto classical Petri Nets for behaviour analysis purposes. Therefore, we consider these as horizontal transformations, since they generate target models (oWFN or Petri Nets) that represent an equivalent behaviour to the source models (BPEL processes), but using a formalism that can be processed by automated tools.

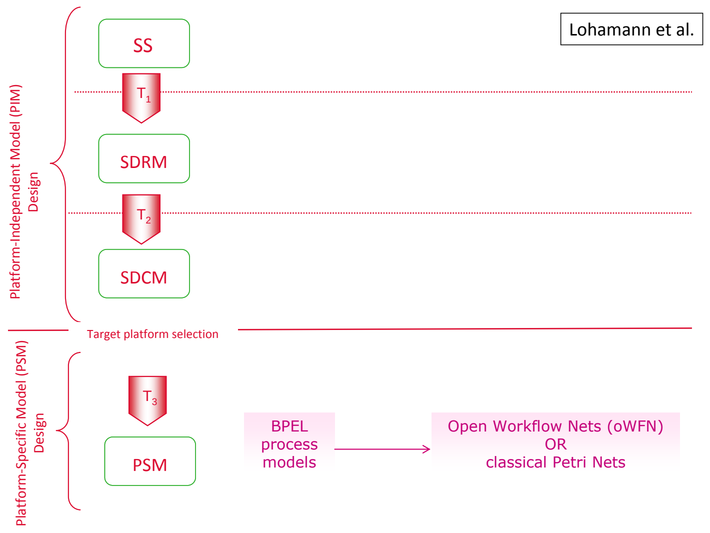

Lohamann et al.
SS
T1
SDRM
T2
SDCM
Platform-Independent Model (PIM) Design
Target platform selection
Platform-Specific Model (PSM) Design
T3
PSM
BPEL process models
Open Workflow Nets (oWFN) OR classical Petri Nets

Figure 54 Positioning of Lohamann et al. with respect to our abstraction levels

4.8 Execution of ISDL Processes using BPEL

The work in [111] proposes a technique for developing a distributed application as a composition of services offered by the application components. In order to realise this composition as an executable orchestration from the perspective of a single component, support for abstraction levels is provided in [111]. Business analysts can specify a service composition as a business process among interacting participants given some business requirements at a high abstraction level. Application designers can create software applications that implement the interactions specified by the business analyst at a lower abstraction level. However, a correctness mechanism is necessary to ensure that implementation levels preserve the behaviour intended at higher abstraction levels. Therefore, [111] proposes a mechanisms of *correctness-by-assessment*, which allows one to build an implementation and check its correctness afterwards against the original requirements. In case this correctness is not (completely) satisfied, the implementation needs to be revisited or rebuilt. This is in contrast with the correctness notion used in this thesis, which is *correctness-by-construction*.

This notion guides the developer in the process of building an implementation using transformation rules that enforce correctness. Therefore, one builds implementations that are correct by construction.

Figure 55 shows the technique used in [111] to transform a service composition model into an executable implementation on a Web Services target implementation platform. The service composition model is represented in ISDL (Interaction System Design Language) [48, 112-113], which is a design language suitable to model distributed systems that allows one to represent behavioural aspects of interacting components. The executable implementation in *Figure 55* is realised in BPEL, which allows one to specify the service composition as an orchestration from the perspective of a coordinator component. This coordinator is realised as a BPEL process, which is exposed to its users as a service provider that offers its services described in WSDL.

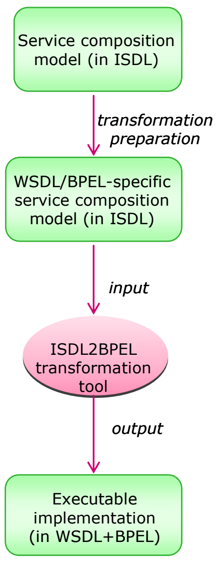

Service composition model (in ISDL)
transformation preparation
WSDL/BPEL-specific service composition model (in ISDL)
input
ISDL2BPEL transformation tool
output
Executable implementation (in WSDL+BPEL)

Figure 55
Transformation technique from a service composition in ISDL to an executable implementation in BPEL

Figure 55 also shows that the ISDL to BPEL transformation is not performed in a single step. First, a *service composition model* in ISDL needs to be manually refined in another ISDL model annotated with some specific WSDL/BPEL information. This refinement prepares the service composition model for the next transformation step by producing a *WSDL/BPEL-specific* service composition model. Afterwards, this model can be automatically transformed by an *ISDL2BPEL* tool into an *executable implementation*, which generates the BPEL process coordinator that orchestartes the service composition, and the WSDL extensions to support this process.

Relevance to this thesis

Figure 56 shows how we have positioned the technique proposed in [111] along the abstraction levels of our methodology. This technique copes with consecutive refinements. A first refinement transforms ISDL abstract interactions among components (partners) into more concrete ISDL interactions, which take into account the internal behaviour of the involved components (partners). Therefore, we positioned the ISDL abstract interactions at our SDRM level and the more concrete ISDL interactions at our SDCM level. A further refinement transforms the ISDL concrete interactions to annotated ISDL interactions that we positioned at the PSM level, since they annotate information specific to the chosen BPEL/Web Services platform. These annotated interactions are finally translated into a BPEL process model and the corresponding WSDL interfaces for execution, which we could also position at the PSM level of our methodology. Therefore, the considered technique proposes a vertical chain of refinements from our SDRM level to the PSM level, as shown in *Figure 56*. Automatic support is provided for the transformation at the PSM level. Concerning our SS level, the ISDL language could be suitable for the purpose of modelling behaviour at this level.

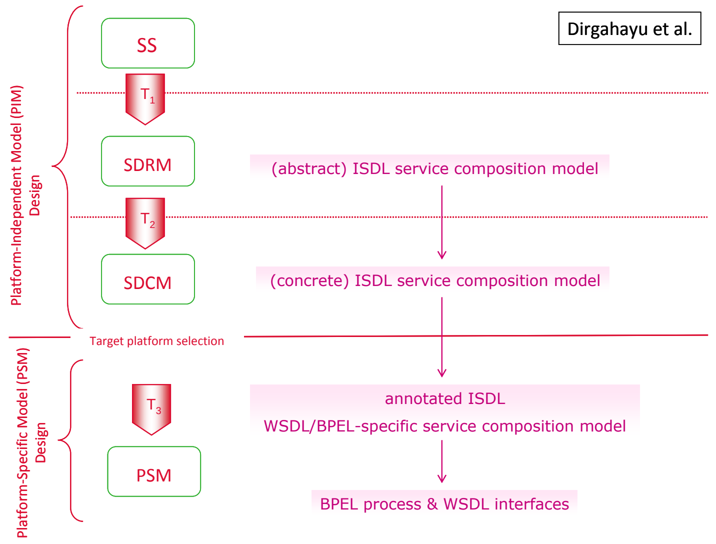

Dirgahayu et al.
SS
T1
SDRM
(abstract) ISDL service composition model
T2
SDCM
(concrete) ISDL service composition model
Platform-Independent Model (PIM) Design
Target platform selection
Platform-Specific Model (PSM) Design
T3
annotated ISDL WSDL/BPEL-specific service composition model
PSM
BPEL process & WSDL interfaces

Figure 56 Positioning of Dirgahayu et al. with respect to our abstraction levels

Chapter 5

Techniques Comparison and Selection

This chapter proposes three behaviour modelling solutions adopted in this thesis to instantiate our methodology. These solutions arise from a comparison among the behaviour modelling techniques discussed in Chapter 4 in the light of some evaluation criteria. These criteria are based on the general concepts and terminology introduced in Chapter 2.

This chapter is organised as follows: Section 5.1 introduces our evaluation criteria, and, based on these criteria, Sections 5.2 to 5.4 discuss the behaviour modelling techniques presented in Chapter 4. Section 5.5 compares these techniques and selects those that are suitable for our methodology. Based on the selected techniques, Section 5.6 proposes three possible solutions that we have experimented with in our research. These solutions are discussed in details in Chapters 6 to 9.

5.1 Evaluation Criteria

In order to instantiate the methodology presented in Chapter 3, we need to prescribe one (or more) language for behaviour modelling at different abstraction levels. We have defined several abstraction levels, from abstract service specifications to executable implementation, each of them with an increasing degree of technical detail. Therefore, several languages can be necessary, ranging from abstract modelling notations to running code. This section presents the criteria[1] we have adopted to evaluate the modelling techniques presented in Chapter 4 in order to select a suitable language(s) for our purpose.

[1]These are qualitative criteria that we have selected based on our experience and relevance to this thesis.

Suitability

Since models always have a purpose (see Section 2.1.2) the language used to represent these models should be suitable for the chosen purpose. Since our models have the purpose to represent the system behaviour, the *suitability* criterion determines whether a certain language gives the means to represent behavioural aspects of the system under development. We have also identified some sub-criteria of suitability, which consists of appeal to intuition and scalability. The *appeal to intution* criterion determines whether the considered language is intuitive to learn and use for system developers. The *scalability* criterion determines whether the considered language provides supports to master complexity of behavioural models when the system under development becomes bigger.

Separation of concerns

The capability of an application design to adapt to possible changes in the technology platform on top of which the application is deployed is a desirable feature in service development. This can be achieved through the separation of application functionality concerns at the PIM level, and technology concerns at the PSM level (see Section 2.1.1). The *separation of concerns* criterion determines whether a certain technique supports this separation of PIM and PSM concerns.

Support for abstraction levels

A design with only one abstraction level would bring either to a model understandable by humans, but with insufficient technical details to be executed by machines, or to a model with all the necessary details to be executed by machines, but hard to understand by humans. Therefore, the decomposition in several abstraction levels that incrementally add technical details towards specific implementations is a desirable feature in a development process (see Section 2.1.6). The *support for abstraction levels* criterion determines whether a certain technique supports this decomposition.

Metamodelling

This thesis focuses on metamodel-based transformations (see Section 2.1.3) and the *metamodelling* criterion addresses the availability of a metamodel of the considered language(s) in order to possibly automate model transformations based on this language(s). The availability of an Ecore version of the metamodel is evaluated positively, since it allows us to execute model transformations in an Eclipse-based environment with tools such as, for example, the ATL and the mediniQVT engines.

Reusabilility
In order to identify best practises that can be reused in other steps of the same development process or, possibly, in the development process of new applications instead of starting from scratch (see Section 2.1.4), the *reusability* criterion determines whether a certain technique support reuse. For example, the explicit usage of patterns is evaluated positively.

Formal support
Although most of the existing modelling languages have a well-defined syntax, these languages often lack formal semantics and, consequently, do not have the basis for automatic verification and validation of behavioural models (see Section 2.1.2). The *formal support* criterion determines whether a formal syntax, both concrete and abstract, and a formal semantics are available for the considered language.

Behaviour correctness
Since systems should behave in the way they are intended to behave, the correctness of models that represent system behaviour should be guaranteed during the development process, possibly already in early stages, by analizing and simulating this behaviour before carrying on with its implementation (see Section 2.1.6). The *behaviour correctness* criterion determines wheteher behaviour analysis, simulation and excution is possible using the considered language.

Tool support
The *tool support* criterion determines whether a certain technique and the adopted language(s) are supported by a proper development environment to represent, transform, simulate, verify and possibly also execute the system behaviour. In other words, this criterion determines what level of automation can be reached by using the considered technique (see Section 2.1.5). Tool support available for public use, easy to get, and well-documented is evaluated positively.

Evaluation approach

In order to evaluate the techniques discussed in Chapter 4 according to the criteria mentioned above, we have combined these criteria in three groups as follows:
– The first group consists of language suitability related criteria, namely the suitability criterion and its appeal to intuition and scalability sub-criteria.

– The second group consists of methodological support related criteria, namely the separation of concerns, support for abstraction levels, metamodelling and reusability criteria.
– The third group consists of automation related criteria, namely the formal support, behaviour correctness and tool support criteria.

For each of these groups we provide a table with the considered criteria in the left side and the techniques under evaluation on top of the table. We evaluate with the symbols • and •• when a given technique addresses a certain criterion partially or completely, respectively. In contrast, the symbol X indicates that the considered technique does not explicitly address or does not provide support for the given criterion[2].

5.2 Language Suitability

Table 1 shows an evaluation of the modelling techniques described in Chapter 4 according to the suitabilty criterion and its sub-criteria, namely appeal to intuition and scalability.

Table 1 Evaluation based on language suitability related criteria

	Uchitel et al.	Harel et al.	Engels et al.	Raedts et al.	Dijkman et al.	Ouyang et al.	Lohmann et al.	Dirgahayu et al.
Suitability	••	••	•	•	••	•	•	••
Appeal to Intuition	X	•	•	•	•	•	X	•
Scalability	X	•	•	••	•	•	••	•

Uchitel et al.

The state machine-based formalism of LTSs and MTSs proposed by Uchitel et al. is highly suitable for behaviour representation purposes, especially to handle synchronization and concurrency issues of interacting components. Therefore, we assign score •• to the suitability criterion. Concerning the appeal to intuition criterion, the synthesis from properties and scenarios technique scores low since it makes use of several modelling notations and formalisms, which requires a lot of expertise since one should be knowledgeable of MSCs, FLTL, LTSs, and MTSs in order to apply this technique. Concerning scalability, state machines-based formalisms, such as

[2]The evaluation of the considered techniques according to our criteria is based on personal judgement.

LTSs and MTSs, are well known to cause state explosion when dealing with large applications. Therefore, we assign score X to the scalability criterion.

Harel et al.

LCSs introduced by Harel et al. (see Section 4.2) support high suitability for behaviour representation. For example, LCSs allow one to distinguish between scenarios that may (possible behaviour) and must happen (required behaviour), and to represent forbidden behaviours and parallel events. Concerning appeal to intuition, LSCs are graphical and follow the sequence diagrams style, which is intuitive for software engineers. However, as stated in [114], LSCs require a learning curve despite the intuitive method for capturing requirements supported by the play-in mechanism, hence, the • score for this criterion. Concerning scalability, a benefit of LSCs is that they support modularity, since they allow one to represent each scenario with a diagram, which can in turn interact with other scenarios/diagrams. However, their drawback is the high number of diagrams running together, which makes large applications difficult to inspect. Therefore, we evaluate scalability as partially supported by LCSs.

Engels et al.

In Engels et al. (see Section 4.3), UML is used as modelling notation for behaviour representation. UML provides a collection of diagrams that can be used to document the design of software systems (descriptive use) or to guide the realisation of these systems (prescriptive use). In its prescriptive use, UML is suitable to generate the structural part of an application by using class diagrams and partial behavioural aspects by combining several types of diagrams, such as use cases, sequence, activity, and object diagrams. This use of several types of diagrams can lead to inconsistencies because of the lack of an unambiguous formal semantics in UML [40]. Therefore, we evaluate the suitability only as partial. The appeal to intuition scores as •, since a benefit of using UML is that it is intuitive and well-known by software engineers. As a drawback, the story-driven technique needs some expertise in coordinating the use of several types of diagrams and notations, including graph rewrite rules and even pieces of Java code in the story diagrams. Concerning scalability, the separation in several types of UML diagrams makes the specification modular, but fragmented as well, especially in case of large applications. Therefore, we evaluate scalability as partially supported.

Raedts et al.

Raedts at al. (see Section 4.4) uses BPMN as modelling notation for business processes. We consider BPMN 2.0 as a suitable notation for representing the system behaviour. However, the proposed technique uses

BPMN 1.0, which lacks some important features, such as the possibility to represent choreography diagrams. Therefore, the suitability criterion is evaluated only as partially supported for this technique. Concerning appeal to intution, BPMN is an intuitive modelling notation, especially for business experts. However, BPMN is used in combination with other formalisms, such as Petri Nets and mCRL2, which are less intuitive to learn and use. Therefore, the appeal to intution criterion scores as • in our evaluation. Concerning scalability, although business process models tend to easily grow out of proportion, BPMN provides mechanisms to collapse elements, which can help reduce model size. Moreover, reduction techniques are used in the framework proposed by Raedts et al. to handle size and complexity of the generated Petri Nets models. Therefore, we assign score •• to the scalability ctiterion.

Dijkman et al.

Dijkman et al. (see Section 4.5) also uses BPMN. As explained above, this is a successful choice in terms of suitability for behaviour representation. However, some parts of the technique proposed by Dijkman et al. still use BPMN 1.0 and some important features cannot be exploited when using this old version. Since more recent work [98] translates and improves the previously achieved results to version 2.0, we evaluate the suitability criterion as fully supported. The appeal to intuition and scalability criteria score both as • because of the benefits of BPMN that we have already discussed above.

Ouyang et al.

Ouyang et al. (see Section 4.6) translates BPMN process models to executable models represented in BPEL. Since this technique applies BPMN version 1.0, we evaluate the suitability for behaviour representation only as partial. The appeal to intuition and scalability criteria score both as • because of the benefits of BPMN that we have already discussed above.

Lohmann et al.

Lohmann et al. defines BPEL semantics in terms of oWFN and Petri Nets (see Section 4.7). We consider BPEL suitable to represent behavioural aspects and concurrency issues, but at implementation level, which is not the scope of this thesis. Although we consider oWFN and Petri Nets suitable for representing concurrent behaviour of distributed systems, we see them more suitable for behaviour analysis purposes than for application requirements modelling purposes. Therefore, we evaluate the suitability of the considered technique as partial. The appeal to intuition criterion scores low, since one has to work at the code level with BPEL, which is not intuitive for non-technical stakeholders, such as business analysts and managers. Concerning the scalability criterion, it scores as fully supported for two reasons: (1) BPEL, analogously to BPMN, provides some way of collapsing elements that can help reducing the model size; (2) the considered technique uses the concept of flexible model generation to minimize the generated oWFN/Petri Nets models during and also after the translation to BPEL. In contrast, standard Petri Nets techniques use tools that help scalability with reduction techniques only after the translation to Petri Nets.

Dirgahayu et al.

Dirgahayu et al. (see Section 4.8) uses ISDL, which is highly suitable to model behavioural aspects in terms of causality relations between interactions. We evaluated the appeal to intuition criterion as partially supported, since the usage of ISDL implies some benefits and drawbacks. The main benefit is that ISDL is a quite intuitive graphical language. However, it still requires a learning curve, which may be not justified because ISDL does not have a widespread adoption, such as, for example, UML or BPMN. Concerning scalability, although the refinements described in [111] seem to grow as the examples become more complex, the ISDL allows one to define composable behaviour modules in order to master the complexity and facilitate the understanding of the resulting behaviour diagrams. Therefore, we evaluated the scalability criterion as partially supported.

5.3 Methodological Support

Table 2 shows an evaluation of the modelling techniques described in Chapter 4 according to the separation of concerns, support for abstraction levels, metamodelling and reusability criteria.

Table 2 Evaluation based on methodological support related criteria

	Uchitel et al.	Harel et al.	Engels et al.	Raedts et al.	Dijkman et al.	Ouyang et al.	Lohmann et al.	Dirgahayu et al.
Separation of concerns	••	••	••	X	••	••	X	••
Support for abstraction levels	•	•	X	X	X	X	X	••
Metamodelling	X	X	••	••	••	••	X	••
Reusability	X	X	X	•	•	••	••	••

Uchitel et al.

We positioned the generation of state transition systems from properties and scenarios at the PIM level of the design process (see Section 4.1). The work in [70] proposes a complementary technique to Uchitel et al. that can be positioned at the PSM level. This technique creates BPEL implementations that can be checked against UML-like abstract specifications. Therefore, the separation of concerns criterion is full supported and we assigned score •• to this criterion. Concerning support for abstraction levels, this criterion scores as • since there is only partial support, namely for our SDRM and SDCM abstraction levels, as shown in *Figure 38*. LTSs and MTSs are more suitable for the specification of detailed behaviours that are already distributed to components, such as our SDCM, and less suitable for the specification of more abstract behaviours, as in the case of our SS. There are no available metamodels, since the transformation from sequence charts to transition systems uses synthesis algorithms implemented in Java. The reusability criterion is not explicitly addressed.

Harel et al.

We positioned the generation of LSCs using the play-in/play-out approach at the PIM level of the design process (see Section 4.2). Although the transformation is still under development, AspectJ code can be generated at the PSM level from LSCs at the PIM level. Alternatively, it is possible to transform these LSCs in state charts, and then generate, for example, Java code. Therefore, the separation of concerns criterion is full supported and we assigned score •• to this criterion. Concerning the support for abstraction levels criterion, our SS and SDRM levels are supported by the play-in and play-out phases of the approach, respectively (see *Figure 41*). However, there is no support for our SDCM level, hence, the support for abstraction levels is partial and this criterion scores as •. The metamodelling and reusability criteria score both as a X. To the best of our knowledge, there is no metamodel provided for LSCs, except for the excerpt in [115] and the attempt in [116], which actually required us a lot of effort to be found. The play-in/play-out approach does not explicitly address the reusability criterion.

Engels et al.

In Engels et al., the separation of concerns criterion scores high since the considered technique clearly distinguishes a modelling phase at the PIM level and a realisation phase at the PSM level. However, there is no support for abstraction levels (see *Figure 44*), since the development process starts directly at the SDCM level by designing a platform-independent model of the architecture that implements the application. The story-driven modelling technique uses UML diagrams and, therefore, the metamodelling criterion is fully supported, since UML metamodels are available. Moreover, excerpts of UML activity and TAAL metamodels used in this technique can be found in [82]. The reusability criterion is not explicitly addressed in the story-driven modelling technique.

Raedts et al.

We considered the model transformations supported by the repository framework in Raedts et al. as horizontal transformations that do no provide separation of PIM and PSM concerns, nor support for abstraction levels (see *Figure 46*). Therefore, both these criteria are not supported and we assigned score X to them. In contrast, metamodelling is fully supported and scores high, since metamodels are available both for BPMN [95] and Petri Nets Markup Language (PNML) that are used in the repository framework. For PNML, an Ecore version is also available [117]. The transformation from BPMN to Petri Nets provides some mechanism for reuse using mapping rules [86], hence, the reusability criterion is (partially) supported.

Dijkman et al.

We considered the two model transformations supported by Dijkman et al. as a horizontal and a vertical transformation, which are realised for behaviour analysis and execution purposes, respectively (see Section 4.5). The horizontal transformation from BPMN to Petri Nets does not contribute to the separation of PIM and PSM concerns or support for abstraction levels either, since it transforms between models at the same abstraction (PIM) level (see *Figure 50*). The vertical transformation to YAWL provides a way to execute at the PSM level the BPMN models that are created at the PIM level, hence, it supports the separation of concerns criterion, which is assigned with score ••. Concerning support for abstraction, the modelling phase starts directly with BPMN process models, which we positioned at our SDCM level (see *Figure 50*), namely the lowest abstraction level of platform-independence in our design. Therefore, abstraction at the PIM level is not supported. Metamodels are available, also in Ecore version, for BPMN, Petri Nets Markup Language (PNML), and YAWL that are used by Dijkman et al. in this technique. Therefore, the metamodelling criterion scores as fully supported. The technique supports reusability, due to the mapping rules from BPMN to Petri Nets provided in [97].

Ouyang et al.

We positioned the transformation from BPMN models to BPEL code supported by Ouyang et al. as a vertical transformation from PIM to PSM levels (see *Figure 52*), which, hence, fully supports the separation of concerns criterion. Abstraction at the PIM level is not supported for the reasons mentioned above, namely the usage of BPMN processes as the only models at the PIM level. Metamodelling is fully supported since Ecore versions of the BPMN and BPEL metamodels are available. Since the technique is entirely developed on patterns for reuse, the reusability criterion is fully supported and scores ••.

Lohmann et al.

In Lohmann et al., the separation of concerns and abstraction levels are not supported, since the proposed transformations focus on the BPEL code at the PSM level without addressing modelling and design concerns at the PIM level. Concerning metamodelling support, the transformation from BPEL to oWFN provides translation algorithms, but a metamodel for oWFN is not provided. Therefore, this criterion scores X. In contrast, reusability is completely supported, since the technique in [107] is based on the translation of BPEL patterns into Petri Nets patterns, which are afterwards composed into an oWFN model.

Dirgahayu et al.

In Dirgahayu et al., the separation of concerns criterion is fully supported since the technique explicitly separates the PIM and PSM levels. Concerning abstraction levels, the ISDL used in this technique can support all our abstraction levels, hence, this criterion scores ••. Metamodelling is supported since Ecore versions for both ISDL and BPEL metamodels are available. The reusability also scores high, since explicitly addressed in the technique by using the so called *interaction pattern refinement* concept [111].

5.4 Automation

Table 3 shows an evaluation of the modelling techniques discussed in Chapter 4 according to the formal support level, behaviour correctness, and tool support criteria.

Table 3 Evaluation based on automation related criteria

	Uchitel et al.	Harel et al.	Engels et al.	Raedts et al.	Dijkman et al.	Ouyang et al.	Lohmann et al.	Dirgahayu et al.
Formal support	••	••	•	••	••	••	••	••
Behaviour correctness	••	••	•	••	••	•	•	•
Tool support	•	•	••	X	•	•	••	•

Uchitel et al.

The formal syntax and semantics for the adopted MSCs diagrams is well defined in terms of LTSs in [118]. Therefore, the formality level is fully supported and scores ••. Concerning behaviour correctness, this technique scores high for several reasons: (1) it supports verification of trace equivalence between two transitions systems generated from an abstract MSC specification and a BPEL implementation, respectively. In this way, it is possible to validate whether the BPEL code satisfies its MSC specifications; (2) it applies model-checking techniques to verify liveness properties and the absence of deadlocks; (3) it allows behaviour simulation and execution using the LTSA and MTSA tools. Concerning tool support, although the technique is supported by the mentioned tools, automatic generation of code at the PSM level is not possible, since the BPEL implementation needs to be manually developed by an expert [70]. Therefore, we evaluated the tool support as partial and assigned score • to this criterion.

Harel et al.

The formal support scores high for LSCs of Harel et al., since the complete LSC syntax and semantics are formally defined in [73, 119]. Further work on LSCs semantics can be found in [120-121], in which mappings of LSCs onto temporal logic are presented. Also behaviour correctness scores high, due to the extensive support for behaviour analysis of LCSs proposed in [115, 122]. Moreover, *Play out* is an excellent mechanisms to simulate the system behaviour at the PIM level, and *smart play-out* [71] provides verification methods, mainly model-checking, to execute and analyse LSCs. Concerning tool support, the play engine is quite an old tool and supports simulation of behaviour but not implementation. Its successor, which is the Eclipse-based tool *playGo*, is currently under development. Therefore, we evaluate tool support only as partial.

Engels et al.

Although the abstract and concrete syntax of UML is well defined in a standard document [22], the UML semantics is only defined in natural language in a fragmented style and sometimes even inconsistently [40]. The story-driven modelling technique by Engels et al. compensates this lack of semantics by defining mappings of UML activity diagrams [123] and TAAL [83] onto transition systems. Since these mappings are limited to UML activity diagrams and TAAL, the formal support is evaluated as partial and scores •. Analogously, behaviour correctness is evaluated as partially supported, since the (limited) UML activity diagrams and TAAL semantics can be used to perform behaviour analysis by using model-checkers, such as Groove. Moreover, behaviour execution at the PIM level is not supported since the story-driven technique mainly focuses on the generation of running Java code. Finally, the tool support scores high. The Fujaba tool supports both code generation and round-trip engineering, while the Groove tool supports validation and verification of behavioural models.

Raedts et al.

Since the repository framework of Raedts et al. focuses on the formalism, analysis and simulation of business process models, it scores high concerning the formal support and behaviour correctness criteria. Although the work in [85-86] claims complete tool support for the transformations, it does not provide any information about where to find these transformation tools. Since we could not find these tools, we assigned X to the tool support criterion.

Dijkman et al.

The formal support of the technique by Dijkman et al. scores high, since the syntax and semantics of BPMN are well defined in [97-98]. Particularly, the BPMN semantics is defined in terms of Petri nets for behaviour analysis purposes, and in terms of YAWL for more advanced behaviour analysis and execution purposes. Therefore, also the behaviour correctness criterion is evaluated as fully supported. Eclipse-based tool support is available in terms of a BPMN to Petri nets transformer, and a BPMN to YAWL transformer [96]. However, these tools do not currently support BPMN version 2.0. Therefore, the tool support criterion is evaluated as partially supported.

Ouyang et al.

The formal support by Ouyang et al. scores high, since it is based on BPMN and BPEL, which have well defined syntax and semantics. Behaviour analysis of BPMN process models is possible by complementing this approach with the transformation from BPMN to Petri Nets realised by Dijkman et al. Moreover, in order to check the correctness of a BPEL process, one should make a reverse transformation from BPEL to BPMN using the patterns in [124], exploit the BPMN to Petri Nets transformation of Dijkman et al., and use the automatic tools for behavioural analysis on these generated Petri nets. Since this mechanism is not straightforward, we evaluated behaviour correctness support (at the PIM level) as partial. The tool support criterion also scores as partial since an Eclipse-based BPMN2BPEL tool is available for automating the transformation, but it currently does not support BPMN version 2.0.

Lohmann et al.

The formal support by Lohmann et al. scores high, since it is based on oWFN, Petri Nets and BPEL, which have well defined syntax and semantics. The tool support criterion also scores high, while behaviour correctness is evaluated as partially supported. This is because the Fiona tool can be used to verify the correctness of the oWFN generated from the source BPEL processes, as well as several automated tools and model checkers can be used for analysis purposes on the classical Petri Nets generated from the same BPEL processes. However, it is not possible to simulate system behaviour at the PIM level before investing in implementation at the PSM level, since the PIM level is not considered in this technique.

Dirgahayu et al.

The full concrete syntax of ISDL used by Dirgahayu et al. is defined in [113], and its semantics is formally defined in [112], hence, the formal support scores high. The behaviour correctness criterion is partially supported since, as discussed in Section 4.8, the technique checks correctness by assessment, namely whether a set of conformance requirements are satisfied, and not by construction, like in the case of this thesis. Moreover, behaviour execution at the PIM level is possible since simulation of ISDL is supported, as discussed in [125]. The tool support scores as partial since, although the work in [111] claims tool availability for transforming ISDL into BPEL, a link to this tool is not provided. The tool could be obtained though after contacting the authors.

5.5 Comparison and Selection

For comparison purposes, *Table 4* shows the analysed behaviour modelling techniques with respect to all our evaluation criteria.

Table 4 Comparison of behaviour modelling techniques

	Uchitel et al.	Harel et al.	Engels et al.	Raedts et al.	Dijkman et al.	Ouyang et al.	Lohmann et al.	Dirgahayu et al.
Suitability	••	••	•	•	••	•	•	••
Appeal to Intuition	X	•	•	•	•	•	X	•
Scalability	X	•	•	••	•	•	••	•
Separation of concerns	••	••	••	X	••	••	X	••
Support for abstraction levels	•	•	X	X	X	X	X	••
Metamodelling	X	X	••	••	••	••	X	••
Reusability	X	X	X	•	•	••	••	••
Formal support	••	••	•	••	••	••	••	••
Behaviour correctness	••	••	•	••	••	•	•	•
Tool support	•	•	••	X	•	•	••	•

Uchitel et al.

The behaviour synthesis from properties and scenarios technique of Uchitel et al. offers an interesting solution, especially due to the high suitability of the employed formalism of transition systems, and the high formal support that can be exploited for behavioural analysis and model checking purposes. Therefore, we selected it as a suitable solution. However, the main problem with this solution consists of the lack of support for the specification of high level abstract behaviours that are not assigned to architectural components, such as our SS models, as shown in *Figure 38*. Therefore, support was necessary to extend this technique in order to cover also our SS level. We present our extension to this technique in Section 5.6.2.

Harel et al.

The play-in/play-out approach of Harel et al. offers a complete solution that provides us with a suitable language (LSCs) for behaviour representation, and support for abstraction levels, including our SS level, as shown in *Figure 41*. Therefore, it is a suitable solution for our purposes, i.e., modelling, simulating, and executing application behaviour in an automatic way, from high level requirements towards final implementations. However, we did not adopt it for the following reasons:

1. There is no metamodel provided for LSCs, which is a main drawback since the focus of this thesis consists of automating model transformations using (Ecore) metamodels.
2. Although the approach is highly automated, we could not exploit the benefits of this automation. At the time of selecting the tool support for our research, the only available tool was the *play engine*, while the more interesting *PlayGo* tool was under development. We believe that the Eclipse-based *PlayGo* tool is a main improvement for the adoption of the play-in/play-out approach by a broader community.

Therefore, we did not use either the formalism of LSCs or the technology of the play engine to represent and realise our models and transformations. However, the following ideas of the play-in/play-out approach have inspired the research carried out in this thesis: (1) the raise of the abstraction level in the specification of the system requirements, which is realised in our SS level, (2) the early execution of the system behaviour for simulation purposes, which is realised in our SDRM and SDCM levels, and (3) the implementation of the expected system behaviour, which is realised in our PSM level, possibly in an automatic manner without the intervention of a technical developer.

Engels et al.

The story-driven modelling technique of Engels et al. is a valuable example of how behavioural aspects of the application under development can be considered already at the PIM level of the design process. However, this technique lacks support for abstraction levels, since the PIM design does not consider behavioural refinements but directly starts the development process by defining a model of the architecture that implements the application (our SDCM level, see *Figure 44*). Moreover, we believe that there are more suitable notations than the employed UML-style diagrams to model the system behaviour at the PIM level [40]. Therefore, we did not adopt the story-driven technique in this thesis.

Raedts et al., Dijkman et al. and Ouyang et al.

The BPMN-based techniques of Raedts et al., Dijkman et al. and Ouyang et al. focus on the transformation of BPMN business process models to equivalent formalisms for behavioural analysis and execution purposes. As shown in *Table 4*, all these techniques lack support for (SS and SDRM) abstraction levels. This had inspired our research, since BPMN 2.0 offers the possibility to specify choreography diagrams that allow one to model the behaviour of participants in business interactions. These choreography diagrams provided us with a means to represent our SS and SDRM levels. Moreover, elements of these BPMN-based tecniques could be beneficially incorporated in our work. For example, the transformations from BPMN to executable languages, such as BPEL (Ouyang et al.) and YAWL (Dijkman et al.), could be exploited to generate executable models at our PSM level, while the transformations from BPMN to Petri Nets (Dijkman et al.) could be used to generate equivalent PIM models for automated behaviour analysis and model checking. Therefore, we selected BPMN as a suitable notation. We discarded the solution proposed by Raedts et al. due to the lack of tool support. We present a solution that uses BPMN in Section 5.6.3.

Lohmann et al.

The work of Lohmann et al. provides a promising solution for verifying that interacting BPEL process models preserve behaviour correctness. However, this solution is limited to the PSM level of our approach, as shown in *Figure 54*. Since our work focuses on PIM level behaviour refinements, this solution does not meet our purposes completely. However, as a partial solution, the BPEL to oWFN transformation could be used in combination with another strategy that allows behaviour modelling at the PIM level, as proposed in Section 5.6.3.

Dirgahayu et al.

The solution proposed by Dirgahayu et al. scores high with respect to all the considered criteria. Therefore, we considered it as a suitable solution that could be used throughout the whole methodology, as illustrated in Section 5.6.1.

5.6 Proposed Solutions

This Section describes how the solutions selected in Section 5.5 could be extended in order to be used in our methodology. *Figure 57* to *Figure 59* show our proposals. In chronological order, we first experimented with ISDL, since it was the language adopted in the context of the A-MUSE project [61], in which the first part of this research was carried out. Since our methodology is language-independent, we applied it also to the synthesis from properties and scenarios technique of Uchitel et al. Moreover, although successful, the choice of ISDL as the only modelling notation for our methodology would have limited our work to a language that is not commonly adopted. Therefore, we used the acquired knowledge to experiment with the BPMN standard, which appears to be gaining popularity in both academia and industry, and is supported by different tools from different vendors.

5.6.1 Behaviour refinements with A-MUSE DSL and ISDL

In parallel to the work of Dirgahayu et al., we have developed a solution that allows the representation of our SS and SDRM levels by using the A-MUSE Domain Specific Language (DSL). The A-MUSE DSL is a profile of ISDL that provides support for representing abstract actions that are not already distributed to individual components or business partners. *Figure 57* shows this solution and highlights our contribution.

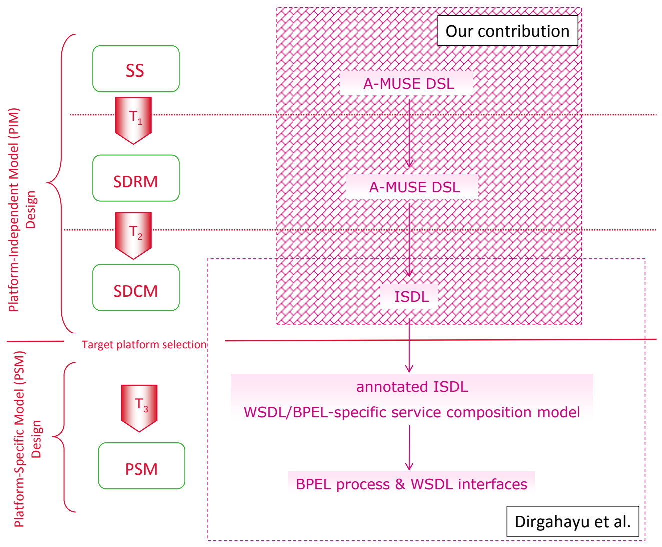

Our contribution
Platform-Independent Model (PIM) Design
SS
T_1
SDRM
T_2
SDCM
A-MUSE DSL
A-MUSE DSL
ISDL
Target platform selection
Platform-Specific Model (PSM) Design
T_3
PSM
annotated ISDL
WSDL/BPEL-specific service composition model
BPEL process & WSDL interfaces
Dirgahayu et al.

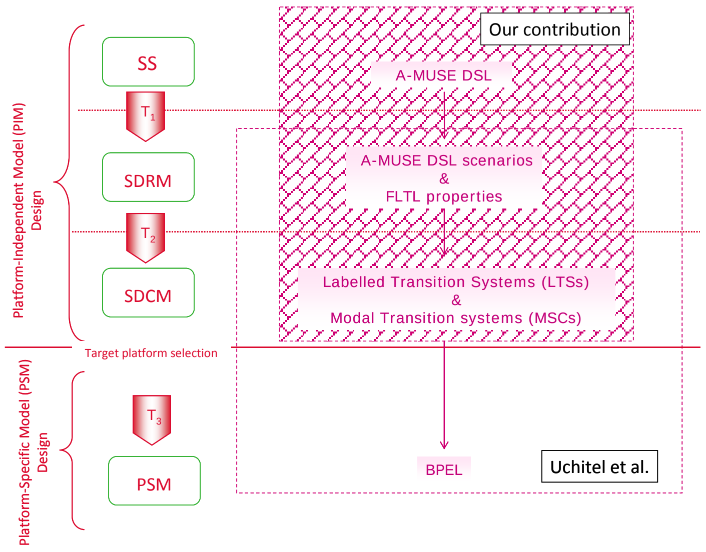
Figure 57 A-MUSE DSL and ISDL solution

An Eclipse-based editor for A-MUSE DSL is available [126] that allowed us to create Ecore versions of our SS and SDRM models. We have used these Ecore models to automatically develop the transformation step T_1 in *Figure* 57. To achieve this aim, we have chosen the medini QVT tool [39], which consists of an Eclipse-based engine that implements the Query/View/Transformation (QVT) Relations standard [24] defined by OMG for mode-to-model transformations. By providing as input to the QVT engine: (1) the A-MUSE DSL metamodel, as both source and target metamodel, (2) the service specification SS as source model, and (3) QVT transformation rules based on interaction patterns to map elements of the SS input model to (more) elements of the SDRM output model, we could automatically generate the service design refined model as target of the transformation. We have also manually realised the second transformation step T_2 in *Figure 57* from A-MUSE DSL to ISDL. Chapter 6 elaborates on this solution.

5.6.2 Synthesis from FLTL properties and A-MUSE DSL scenarios

Since the transformation from SS to SDRM with the Medini QVT engine was proven to be successful [127], we re-used the same strategy to extend the approach by Uchitel et al.. *Figure 58* shows this solution and highlights our contribution.

Figure 58 A-MUSE DSL and TSs solution



Platform-Independent Model (PIM) Design
SS
T1
SDRM
T2
SDCM
Target platform selection
Platform-Specific Model (PSM) Design
T3
PSM
Our contribution
A-MUSE DSL
A-MUSE DSL scenarios & FLTL properties
Labelled Transition Systems (LTSs) & Modal Transition systems (MSCs)
BPEL
Uchitel et al.

As depicted in *Figure 58*, the SS is represented using the A-MUSE DSL and can be used to automatically generate an A-MUSE DSL model at the SDRM level. This model represents our scenario and has been extended with the specification of some safety properties in FLTL, which is able to represent constraints that would not be possible to express with the A-MUSE DSL notation [128]. By adapting the technique in [67], we have synthesized an LTS from our A-MUSE DSL scenarios. Moreover, using the algorithm in [64], we have also synthesized an LTS from FLTL safety properties. Following the Uchitel et al. technique, we have further synthesized these LTSs into corresponding MTSs and, finally, merged these two MTSs in one MTS from properties and scenarios. All the mentioned synthesis steps were performed manually. Chapter 7 elaborates on this solution.

5.6.3 From BPMN choreographies to BPMN orchestrations

In the literature one finds many attempts to develop transformations from PIM business process models to some other formalisms, such as Petri nets and process algebras, or to some PSM implementations, for example, in terms of BPEL. Therefore, we decided to experiment with a process-based approach as an alternative to the transition systems-based tecnique used in Section 5.6.2. *Figure 59* shows the proposed solution and highlights our contribution.

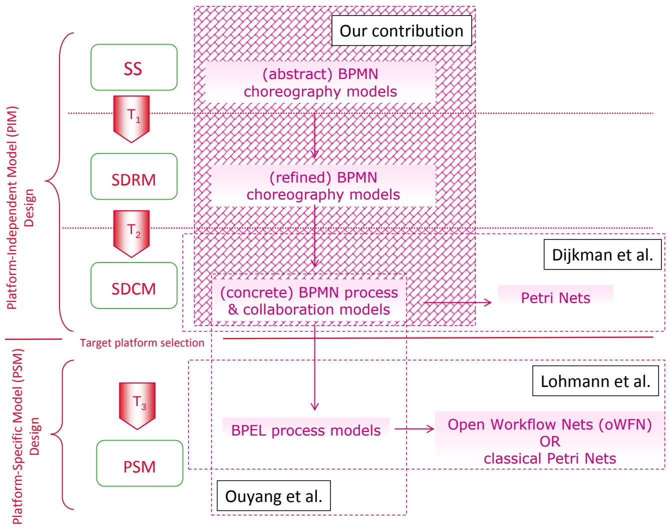

Our contribution
(abstract) BPMN choreography models
(refined) BPMN choreography models
(concrete) BPMN process & collaboration models
Dijkman et al.
Petri Nets
Lohmann et al.
BPEL process models
Open Workflow Nets (oWFN) OR classical Petri Nets
Ouyang et al.
SS
T1
SDRM
T2
SDCM
T3
PSM
Target platform selection
Platform-Independent Model (PIM) Design
Platform-Specific Model (PSM) Design

Figure 59 BPMN solution

As depicted in *Figure 59*, we have used BPMN to model the PIM design of our methodology. We have created the SS in *Figure 59* as a BPMN choreography diagram that describes the abstract interactions between the system and its users, the SDRM as a more detailed BPMN choreography diagram that represents the interactions among the system components, and the SDCM as a collaboration of process models (orchestration) that conforms to the choreography previously defined in the SDRM. We have used the Eclipse-based ATL engine to execute the two PIM model transformation steps in *Figure 59*, namely T_1 from SS to SDRM, and T_2 from SDRM to SDCM. These transformations are defined in ATL and, analogously to the QVT transformations mentioned in Section 5.6.1, these transformations are based on interaction patterns. In this way, we could realise automatic behavioural refinements, enforcing reuse and using a single language, namely BPMN, throughout the whole PIM design process. Chapter 8 elaborates on this solution.

Chapter 6

Behaviour Refinement using A-MUSE DSL and ISDL

This chapter has two purposes, namely (1) to introduce the Live Contacts application, which is the context-aware mobile application that we have used as running example to develop the three solutions outlined in Chapter 5, and (2) to present the first of these three solutions, i.e., a behaviour refinement and synthesis technique that uses A-MUSE DSL and ISDL as modelling languages at the PIM level of our methodology. In order to achieve this, the chapter discusses the source and target models of our PIM behaviour refinement and synthesis transformations, and presents these transformations as well. The *service specification (SS)* and *service design refined model (SDRM)* are the source and target models of our first transformation, namely the *SStoSDRM refinement transformation*. The *service design refined model (SDRM)* and the *service design component model (SDCM)* are the source and target models of our second transformation, namely the *SDRMtoSDCM synthesis transformation*. The SS and SDRM models are represented using A-MUSE DSL, while the SDCM model is represented using ISDL. The chapter focuses on the implantation of the *SStoSDRM refinement transformation*.

This chapter is organised as follows: Section 6.1 introduces the functions offered by the Live Contacts application running example, together with the UML information and context models that represent the (context) information handled by these functions, Sections 6.2 and 6.3 present the SS and SDRM models, respectively, using the Live Contacts running example, Section 6.4 discusses the SS to SDRM transformation, Section 6.5 discusses the SDCM model and, finally, Section 6.6 presents our conclusions about the experience with A-MUSE DSL and ISDL modelling languages.

6.1 Running Example: Live Contacts

The Live Contacts application [60] has been originally developed in the Business4Users (B4U) project [129] and applied afterwards as an example application in the A-MUSE project [61]. Live Contacts consists of an application that offers context-aware mobile services to its users in order to contact the right person, at the right time, via the right communication channel. Live Contacts has been conceived according to empirical research on the strategies that employees use to reach each other in their working environment [130]. We have used the Live Contacts application as running example to illustrate our methodology, but this application is not the goal of this thesis. Live Contacts is a suitable example for our purpose for the following reasons: (1) it is a means to experiment with traditional request/response aspects of application behaviour, but also with event-based aspects typical of context-aware mobile applications, (2) it is not too simple to become a trivial example, and (3) it is not too complex to become an unmanageable example. *Table 5* summarises the service functions offered by the Live Contacts application.

Table 5 Live Contacts functions

Function name	Function purpose
Sign in	Access to Live Contacts session
Buddy list	Request of live contacts' list of user
Buddy status	Request of IM status of a specific live contact
Add buddy	Addition of new live contact to user's list
Remove buddy	Removal of existing live contact from user's list
Contact buddy	Opening of communication channel bewteen user and live contact
Reminder	Reminder for scheduled user's activity
Status change	Notification of live contact's IM status change
Proximity	Notification of live contact's proximity to user's location
Sign out	Exit from Live Contacts session

In order to interact with his live contacts in the application, a user must sign in and this creates a live contacts session. Afterwards, the user can generate user input events, which require his explicit intervention, or get notifications of context events, which are generated by the application according to the user's context, preferences and needs. Examples of user input events in *Table 5* consist of *buddy list*, *buddy status*, *add buddy*, *remove buddy*, and *reminder* functions. These functions allow a user to request the list of his live contacts (buddies), get information about the IM (Instant Messaging) status of these buddies, add new buddies to the live contacts' list, remove an existing buddy, and set a reminder for scheduled activities, respectively. In *Table 5*, the *status change* and *proximity event* functions are context events, which allow a user to get automatic notifications when a buddy changes his IM status in the application, and a buddy whose IM status is online comes nearby the user, respectively. The *contact buddy* function in *Table 5* connects the user via a defined communication means with a specific live contact. This communication means can be a telephone call, an SMS, an IM service or e-mail.

6.1.1 Information Model

Figure 60 shows the information model of the Live Contacts application. This model is represented as an UML class diagram and describes the data and status information handled by the Live Contacts application.

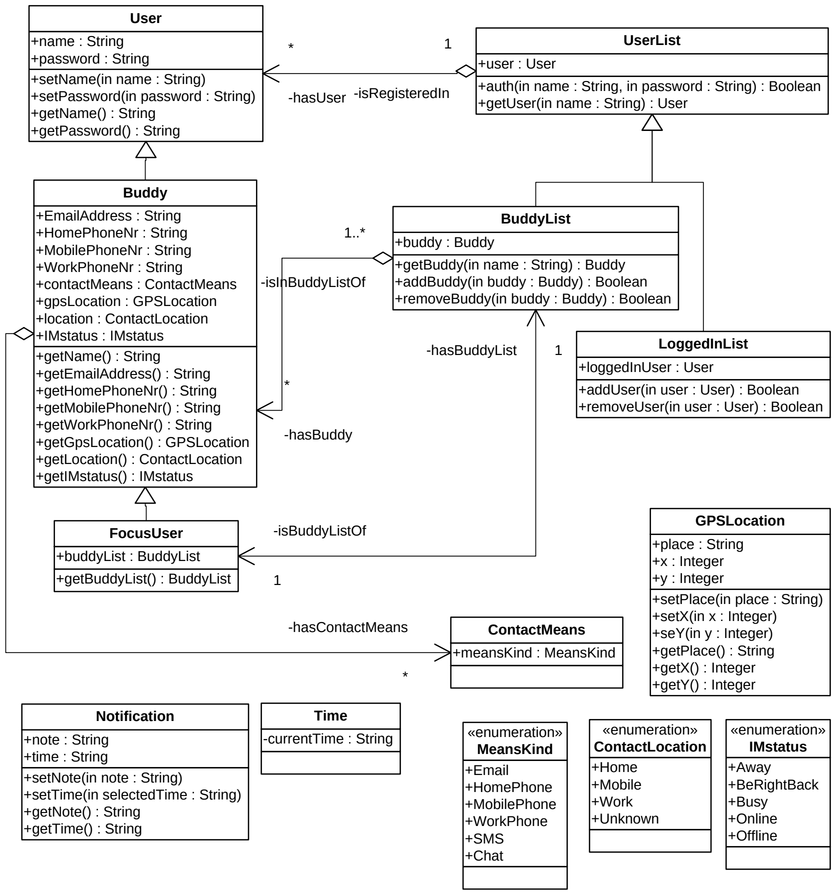


Figure 60 UML information model of the Live Contacts running example

A User represents somebody registered in the Live Contacts application with a name and password. The UserList class represents the set of users registered in the application. The LoggedInList class, which is a subclass of UserList, represents all the registered users that are currently logged in and, therefore, can use the services offered by the application.

The FocusUser class depicted in *Figure 60* represents the user from whose perspective the application is considered. This FocusUser has a BuddyList, which is a subclass of UserList and represents all the buddies of the focus user. The Buddy class provides detailed information about buddies, such as their EmailAddress, PhoneNr, ContactMeans, ContactLocation, GPSlocation and IMstatus. The application uses this information to offer its services to the focus user. The contact location is used to select an appropriate communication channel to contact a buddy. For example, the SMS option should be preferred if the contact's location is set on "mobile" and, consequently, the considered buddy cannot be reached by chat or fixed phone. The GPS location is used to provide context events, such as, for example, the proximity event when a buddy is nearby the user. Both Buddy and FocusUser are users registered in the application. However we have defined FocusUser as subclass of Buddy, since the focus user is also a buddy for other users. The information model of *Figure 60* also shows the Notification and Time classes, which allow the users to set reminders for a specific time.

6.1.2 Context model

Figure 61 shows an excerpt of the context model of the Live Contacts application. This model is represented as an UML class diagram, which describes the relevant concepts handled by the components that manipulate context.

Figure 61 shows the Entity and Context classes, which are foundation concepts in our context models. An entity, for example Person, may be related to several different context aspects, such as, for example, Location and Temperature (see Section 2.3.2). In contrast, a specific context aspect may relate to one or more entities. For example, the Location context aspect relates to multiple entities, such as Person and Device. The SpatialEntity class in *Figure 61* represents tangible objects, such as a person or a device. In contrast, an intangible entity represents intangible objects, such as, for example, an application or a network. The IntrinsicContext class in *Figure 61* represents a type of context aspects that belongs to the essential nature of a single entity and does not depend on the relationship with other entities [50]. Examples of this type of context aspects are the location of a person or a device. The FocusUser and Buddy classes in *Figure 61* represent user types or roles in the Live Contacts application and are related to each other according to the information model in *Figure 60*.

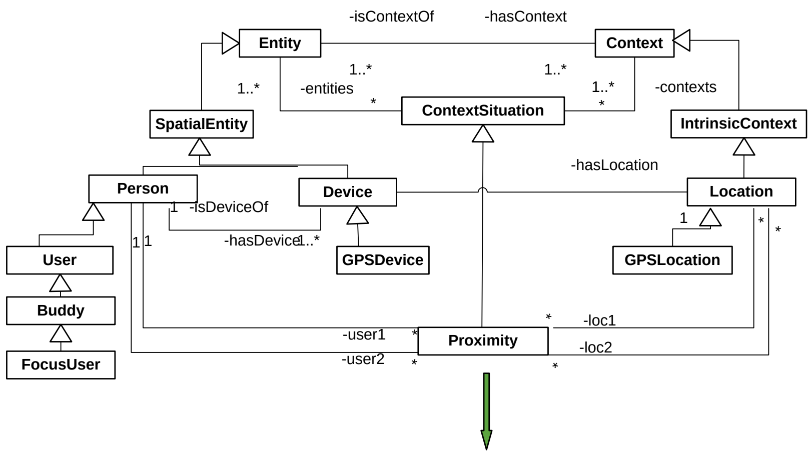


Figure 61 Context model for the proximity event

```
Proximity
user1 = FocusUser;
user2 = Buddy;
loc1 = user1.hasDevice(GPSDevice).hasLocation(GPSLocation);
loc2 = user2.hasDevice(GPSDevice).hasLocation(GPSLocation);
distance (loc1, loc2) =
{6378*acos(sin(loc1.x)*sin(loc2.x)+cos(loc1.x)*cos(loc2.x)*cos(loc1.y-loc2.y))};
proximity (user1, user2, threshold) = EVAL {distance (loc1, loc2) < threshold};
```

Figure 61 also depicts the ContextSituation class, which is an element composed by contexts and entities. Context situations enable the representation of particular state-of-affairs of the applications' universe of discourse [50]. An example of context situations is Proximity in *Figure 61*, which describes when a focus user is nearby one of his buddies. The concepts used in the proximity situation can be navigated using the enclosed textual description in *Figure 61*. Particularly, the proximity situation involves two entities of type Person, namely user1 and user2, and two context aspects of type Location, namely loc1 and loc2. The entity user1 corresponds to the focus user (user1 = FocusUser), while the entity user2 corresponds toone of the buddies of this focus user (user2 = Buddy). By navigating *Figure 61* from the Proximity class to the left, we can follow the condition loc1 = user1.hasDevice(GPSDevice).hasLocation(GPSLocation), i.e., the user1 is a Person, who has one (or more) Device of type GPSDevice, which has only one Location of type GPSLocation. This GPS Location class is further represented in the information model of *Figure 60*. Loc1 is an intrinsic context type in *Figure 61* and an element of the Proximity situation. This situation compares the location of two persons based on the distance between their locations, and evaluates to true when this distance is within a certain threshold, i.e., proximity (user1, user2, threshold) = EVAL {distance (loc1, loc2) < threshold}. The distance (loc1, loc2) method in *Figure 61* uses the x and y attributes of the GPSLocation class defined in the information model of *Figure 60*.

6.1.3 Behaviour models

The information and context model described in Sections 6.1.1 and 6.1.2, respectively, apply at different abstraction levels of our methodology, i.e., the same information (context) model can be used at the SS, SDRM and SDCM levels. In contrast, the models that describe the behaviour of the application apply at a specific abstraction level of our methodology. Therefore, different behaviour models are necessary at the SS, SDRM and SDCM levels. These behaviour models gradually add details starting from an abstract SS, going through a partially refined SDRM model, and ending in a possibly executable SDCM model. The next Sections describe these behaviour models using the A-MUSE DSL for the SS and SDRM models and ISDL for the SDCM model. These models are illustrated using the Live Contacts running example and manipulate the information and context information represented in *Figure 60* and *Figure 61*, respectively.

6.2 Service Specification

The service specification (SS) is the most abstract model of our methodology and represents the application to be developed as a single entity with its behaviour being defined from an integrated perspective (see Section 2.2.2). According to this integrated perspective, only the interactions between the system, considered as a black box, and its user, which forms the external environment to the system, are relevant at the SS level.

6.2.1 High-level structure

Figure 62 shows the high-level structure of the service specification of the Live Contacts running example expressed in the A-MUSE Domain Specific Language (DSL) [126]. The A-MUSE DSL is a profile of ISDL that has been developed and applied in the A-MUSE project [61]. An Ecore version of the A-MUSE metamodel is available, together with an Eclipse-based editor to create A-MUSE DSL models that conform to this metamodel.

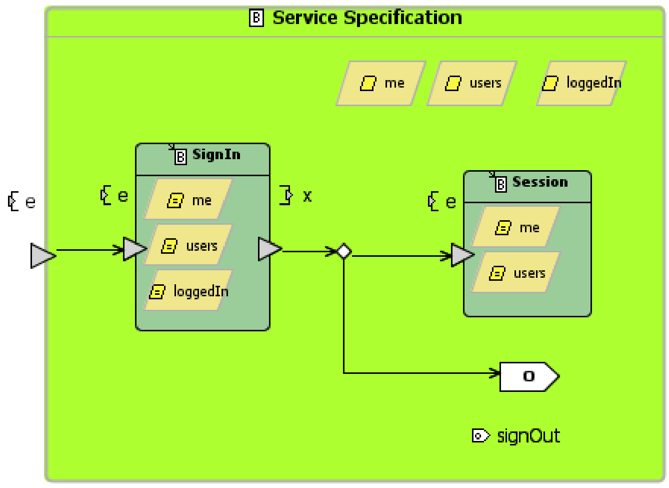


Figure 62 Service Specification (SS) in A-MUSE DSL

Figure 62 represents an instance of the *behaviour* element of the A-MUSE DSL metamodel named Service Specification, which starts with an *entry point* element named e. *Figure 62* also shows three *item* elements named me, users and loggedIn, which are of the types FocusUser, UserList and LoggedInList of the information model in *Figure 60*. Items are global variables that can be referred to within the behaviour that defines these variables. Therefore, the items me, users and loggedIn can be referred to within the Service Specification behaviour. *Figure 63* also shows that in order to access the functions offered by the Live Contacts application, a user must first request the Sign In function. Afterwards, the user can decide to start a Session or eventually to Sign Out and exit the application. The Sign In and Session are *behaviour instance* elements, which are instances of Sign In and Session behaviour elements described elsewhere in the model. The Session behaviour element is shown in *Figure 63*.

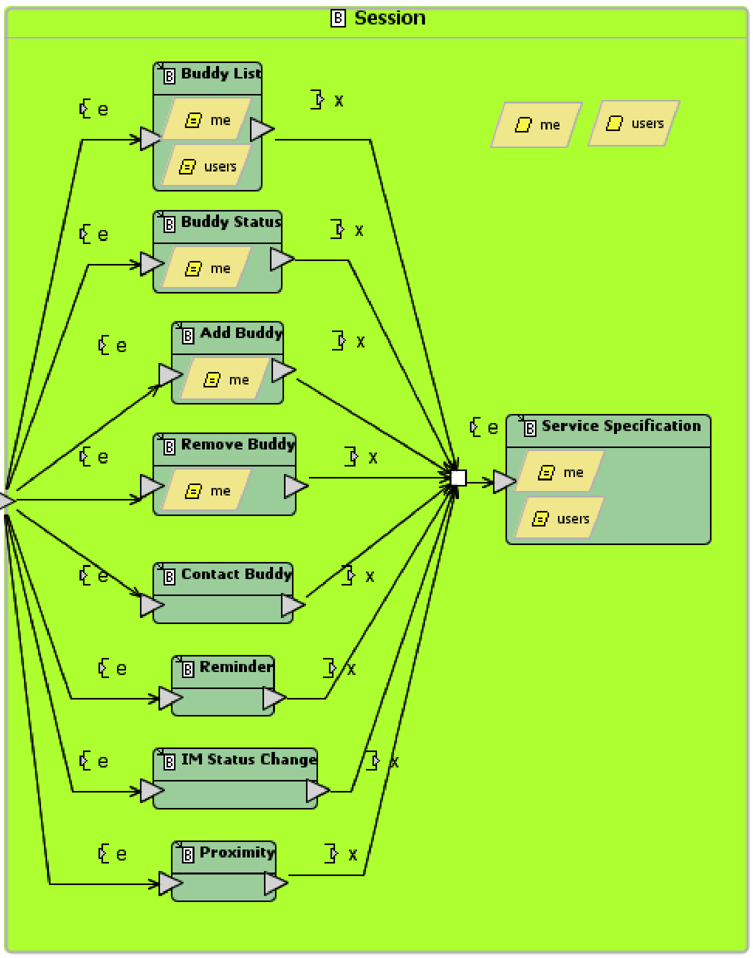


Figure 63 SS, Session

The Session behaviour in *Figure 63* allows alternative options, which correspond to the functions offered by the Live Contacts application shown in *Table 5*. When one of these functions ends, a new instance of the Service Specification behaviour is started and a new option can be chosen. Each function is represented in *Figure 63* as a *behaviour instance* element, which belongs to a behaviour element represented in another module of the specification. In the remainder of this thesis, we use the Remove Buddy and Proximity functions instead of the whole behaviour, since these functions are representative of traditional request/response interactions between the user and the system (Remove Buddy function), and event-based interactions that do not require explicit user intervention (Proximity function).

6.2.2 Service functions

Figure 64 and *Figure 65* zoom into the details of the Remove Buddy and Proximity behaviour instance elements (functions), respectively.

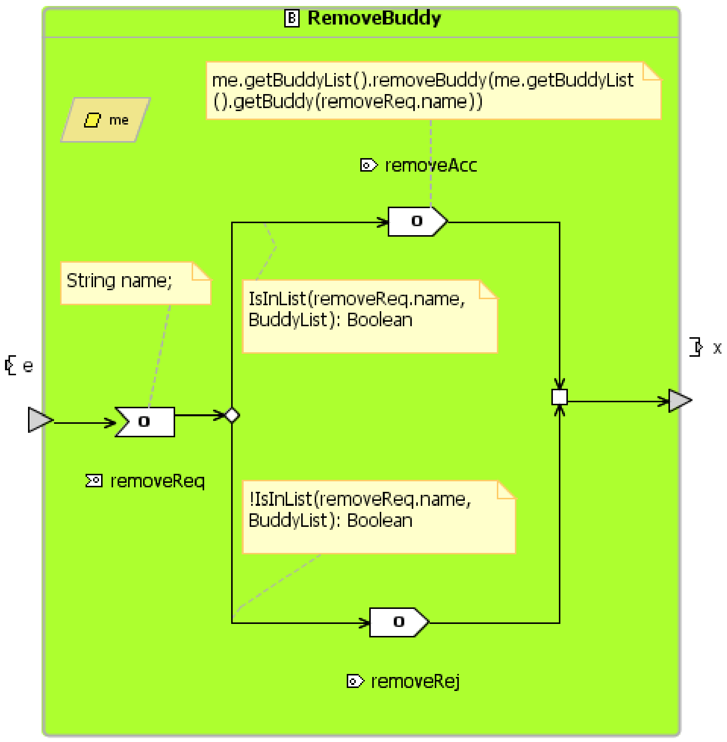
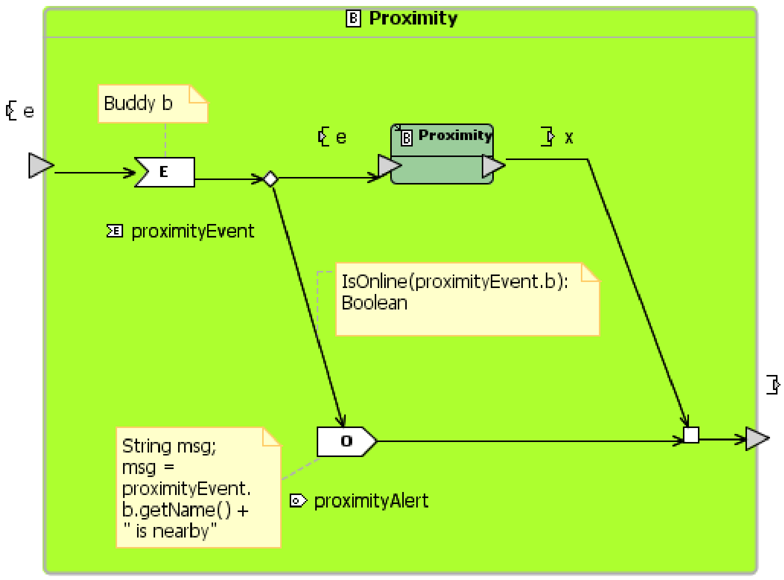

RemoveBuddy
me
me.getBuddyList().removeBuddy(me.getBuddyList().getBuddy(removeReq.name))
removeAcc
o
String name;
IsInList(removeReq.name, BuddyList): Boolean
e
x
o
removeReq
!IsInList(removeReq.name, BuddyList): Boolean
o
removeRej

Figure 64 SS, Remove Buddy

The Remove Buddy function (behaviour element) in *Figure 64* consists of a *user input* element named removeReq followed either by a *user output* element named removeAcc or a *user output* element named removeRej. We use the *user input* and *output* elements as *interaction markers* (see Section 3.2), namely as placeholders for abstract interactions at the SS level that correspond to (more concrete) refined interactions among specific components at the SDRM level.

Figure 64 shows that a user may request to remove a buddy from his buddy list (removeReq marker) by giving as input to the application the name of this buddy (String name). If the buddy is not in the list (!IsInList(removeReq.name, BuddyList): Boolean condition), the user request is rejected (removeRej marker), otherwise (IsInList(removeReq.name, BuddyList): Boolean condition) the request is accepted (removeAcc marker) and the buddy is removed from the buddy list of the user (represented by the me.getBuddyList().removeBuddy(me.getBuddyList().getBuddy(removeReq.name)) method). The status information handled by the Remove Buddy function is defined in the UML information model in *Figure 60*. Although this information is represented in *Figure 64* using textual annotations attached to model elements, this is done only for representation purposes, since the Eclipse-based A-MUSE DSL editor [126] allows one to specify this information in a property view separately from the graphical representation of model elements.

Figure 65 SS, Proximity



The Proximity function (behaviour element) consists of an *event* element named proximityEvent followed either by a *user output* element named proximityAlert or a new instance of the Proximity behaviour. We use these *event* and *user output* elements as interaction markers for the Proximity function.

Figure 65 shows that a user can be notified about the occurrence of the proximity situation represented in our context model (see Section 6.1.2). The occurrence of a proximity situation is represented in *Figure 65* with the proximityEvent marker when a buddy (Buddy b), whose IM status is “online” (IsOnline(proximityEvent.b): Boolean condition) is nearby the user. As a consequence, the application warns the user with an alert (proximityAlert marker). The specific text shown to the user, namely the message msg = proximityEvent.b.getName() + “is nearby”, it is not relevant at this level of abstraction, but is represented for consistency with the service design refined level.

6.3 Service Design Refined Model

The service design refined model (SDRM) represents the application to be developed as a structured behaviour from a distributed perspective (see Section 2.2.2). According to this distributed perspective, the system is considered as a set of interacting components with interfaces that offer services to each other, independently on the specific internal behaviour of each component. 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 thesis, we used the reference architecture for context-aware mobile applications presented in Section 3.4 and recalled in *Figure 120*.

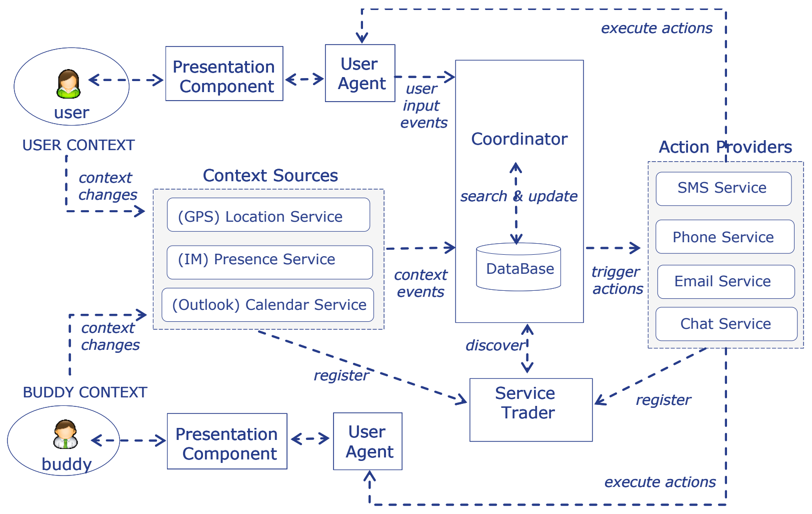


Figure 66 Reference architecture for context-aware mobile applications

6.3.1 Remove Buddy refinement

Figure 67 shows the Remove Buddy behaviour that refines the service specification in *Figure 64* in terms of interactions between components of our reference architecture. This behaviour involves the user agent (UA), coordinator (C) and database (DB) components. Each interaction in *Figure 67* is marked with a label and represents an interaction between two components of our reference architecture 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 *Figure 64*, but assigned to the proper corresponding refined interactions.

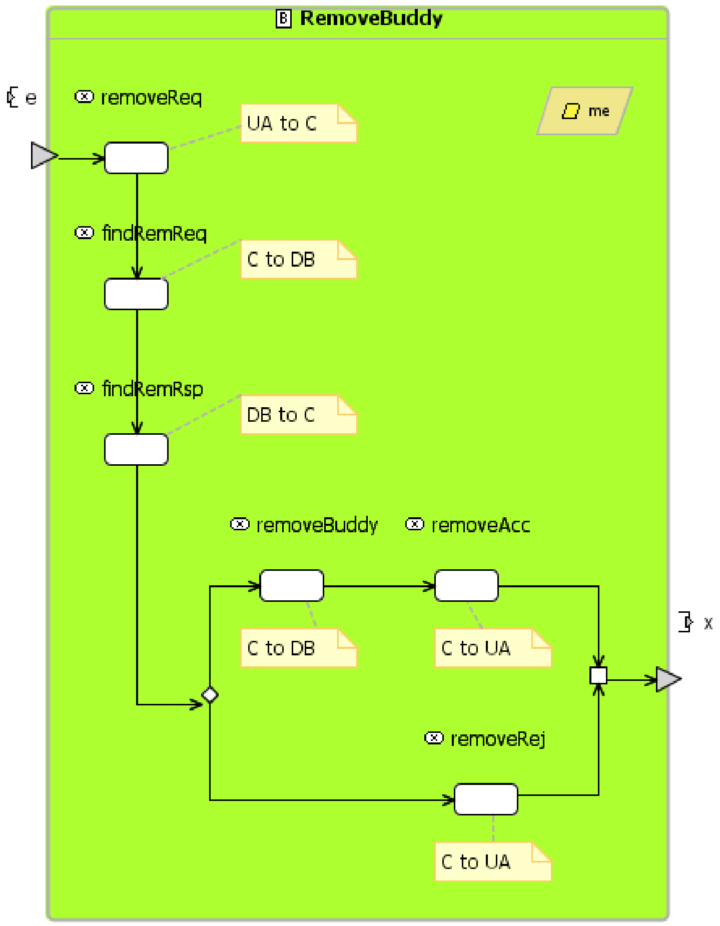


Figure 67 SDRM, Remove Buddy

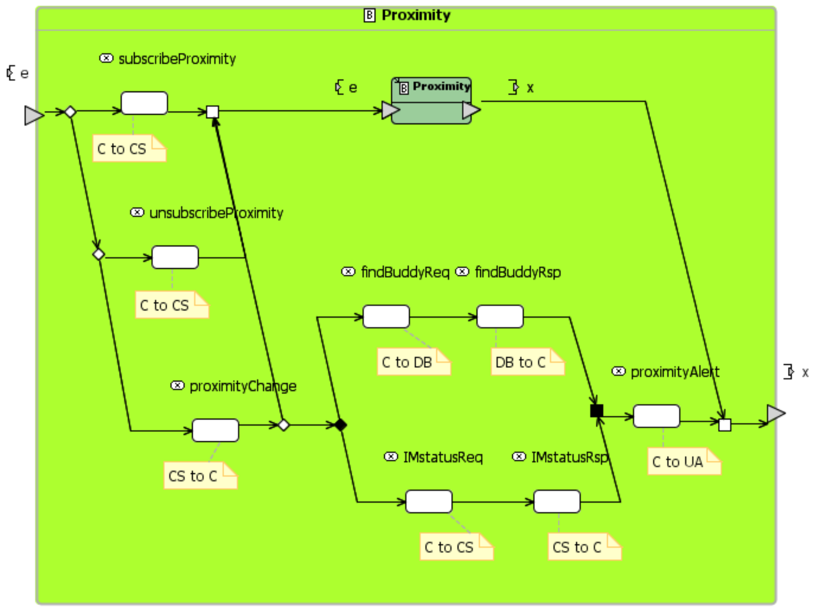
The removeReq interaction in *Figure* 67 consists of a request from the user agent to the coordinator (UA to C label) to remove a buddy from the user's list. The coordinator interacts with the database through the findRemReq and findRemRsp interactions (C to DB and DB to C labels, respectively) to determine whether the buddy is included in the buddy list of the user. If this is the case, the coordinator removes the buddy from the list with the removeBuddy interaction (C to DB label) and sends a positive response to the user agent through the removeAcc interaction (C to UA label). If the buddy is not in the list, the coordinator sends a negative response to the user agent through the removeRej interaction (C to UA label).

In *Figure* 67, we identified the following five *basic interaction patterns* (see Section 3.2), which are recurrent interactions between two components: (1) the *request* pattern consists of a one-way interaction between the user agent and the coordinator (removeReq in *Figure* 67), (2) the *search* pattern consists of a two-way interaction between the coordinator and the database (findRemReq and findRemRsp in *Figure 67*), (3) the *update* pattern consists of a one-way interaction between the coordinator and the database (removeBuddy in *Figure 67*), (4) the *acceptance* pattern consists of a one-way interaction between the coordinator and the user agent (removeAcc in *Figure 67*), and (5) the *rejection* pattern consists of a one-way interaction between the coordinator and the user agent (removeRej in *Figure 67*).

6.3.2 Proximity refinement

Figure 68 shows the Proximity behaviour that refines the service specification in *Figure 65* in terms of interactions between components of our reference architecture. This behaviour involves the coordinator (C), context sources (CS), database (DB) and user agent (UA) components. Context sources (CS) are the components dedicated to sense changes in the user's context and provide the coordinator (C) with context events.

Figure 68 SDRM, Proximity



Although there are several context sources distributed in the environment, we assume in *Figure 68* that only one context source at a time communicates with the coordinator, namely the context source that has sensed the event of interest. In case of the proximity event, we assume that this context source consists of a context manager component (see Section 2.3.1) that:

- has access to raw context information, namely the user and buddies location coordinates captured by single domain sensors, such as, for example, the GPS devices integrated in the user and buddies mobile phones,
- combines this raw information coming from multiple sources in aggregated information, namely the proximity situation discussed in Section 6.1.2, and
- generates a proximity event upon the occurrence of the proximity situation between the user and one of his/her buddies.

In order to receive context events, it is first necessary to subscribe to those events. *Figure 68* shows the subscribeProximity interaction between the coordinator and a context source (C to CS label). After the subscription, a proximityChange is generated eventually by this context source to notify the coordinator upon the occurrence of a proximityChange (CS to C label). *Figure 68* further shows that the proximityChange is followed by a parallel decision with two branches, which merge after the following has taken place: (1) retrieving from the database the name of the buddy of interest with the findBuddyReq and findBuddyRsp interactions (C to DB and DB to C labels, respectively), and (2) querying an appropriate context source in order to synchronously retrieve the IM status of this buddy with the IMstatusReq and IMstatusRsp interactions (C to CS and CS to C labels, respectively). In case the retrieved IM status is "online" the proximityAlert interaction occurs, in which the coordinator generates a message to notify the occurrence of a proximity event to the user agent. If the buddy is not "online", no user alert is generated. After the subscribeProximity interaction, proximityChange events can be generated by the context source as long as the unsubscribeProximity interaction depicted in *Figure 68* does not occur.

The context subscription mechanism represented in *Figure 68* is controlled by a variable with name Subscribed. The subscribeProximity interaction can occur when the mentioned variable is set on "!Subscribed". Once the subscription is done, the control variable must be set on "Subscribed", in which case two events can occur: (1) the proximityUnsubscribe interaction in case the Coordinator is not anymore interested in proximity notifications from the Context Source, after which the control variable must be set on "!Subscribed", or (2) a proximityChange interaction generated eventually by the Context Source to notify the Coordinator upon the occurrence of a proximity event, after which the control variable keeps the value "Subscribed". The actual implementation of the context subscription and notification mechanisms falls outside the scope of this thesis. However, in our previous work we have tackled how a context expression evaluator component dedicated to context information sensing and processing can be used in our reference architecture for these purposes [131].

The refinement shown in *Figure 68* is one of the possible solutions that can be used to define the Proximity function, possibly the most straightforward solution. However, other solutions can be used. In our research approach (see Section 3.2), we have designed this refinement manually and, afterwards, used the knowledge generated in the design to automate this refinement. In the phase of manual refinement, we identified the following six basic interaction patterns in *Figure 68*: (1) a one-way *subscribe* pattern between coordinator and context source (subscribeProximity in *Figure 68*), (2) a one-way *unsubscribe* pattern between coordinator and context source (unsubscribeProximity in *Figure 68*), (3) a one-way *signal event* pattern between context source and coordinator (proximityChange in *Figure 68*), (4) a two-way *search* pattern between coordinator and database (findBuddyReq and findBuddyRsp in *Figure 68*), (5) a two-way *context query* pattern between coordinator and context source (IMstatusReq and IMstatusRsp in *Figure 68*), and (6) a one-way *event alert* pattern between coordinator and user agent (proximityAlert in *Figure 68*).

6.4 SS to SDRM Refinement Transformation

In order to automatically generate the SDRM target model of *Figure 67* from the SS source model of *Figure 62*, we have created a transformation called *SStoSDRM refinement* that is based on interaction markers and patterns as units of reuse. This transformation consists of transformation rules in the QVT Relations language supported by the Medini QVT tool [39]. Inputs to the Medini QVT transformation are:

- a source and a target metamodel defined in *Ecore*, which is the metamodel type used by the Eclipse Modeling Framework (EMF) [26]. Our source and target metamodels are both represented by the A-MUSE DSL metamodel,
- a source model conforming to the source metamodel, which is the SS expressed in A-MUSE DSL.

The Medini QVT transformation produces as output a target model that conforms to the given target metamodel, namely an SDRM model expressed in A-MUSE DSL. The next Sections show schematically the QVT transformation rules that realise the SS to SDRM refinement of the Remove Buddy and Proximity functions.

6.4.1 Remove Buddy refinement transformation

Figure 69 shows the source and target models for the transformation of the Remove Buddy service specification discussed in Section 6.2 into the Remove Buddy service design refined model discussed in Section 6.3.

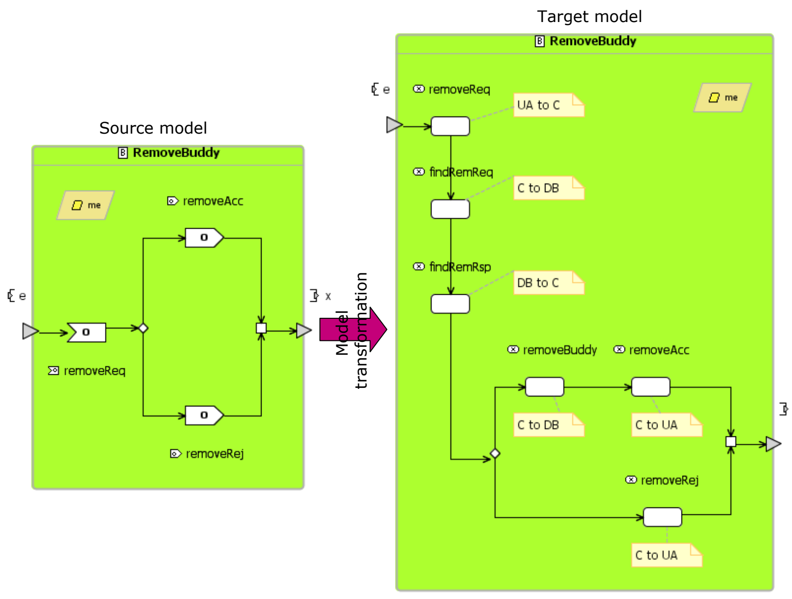


Figure 69 Remove Buddy: source and target models for the SStoSDRM transformation

Figure 69 shows at a glance that the Remove Buddy refinement transformation adds detail to the target model and, at the same time, preserves the behaviour structure of the source model. We have defined transformation rules in order to map interaction markers in the SS source model onto interaction patterns in the SDRM target model. We have also defined transformation rules in order to map the SS behaviour structure, such as, for example, *entry point*, *enabling relation*, *or-split*, *or-join* and *entry point* elements in *Figure 69*, onto corresponding behaviour structure elements in the SDRM target model. For sake of readability, we could not present the complete set of transformation rules. *Figure 70* shows the mappings that we have used to define the transformation rules from interaction markers to interaction patterns, which we consider the most significant transformation rules for the purpose of this thesis.

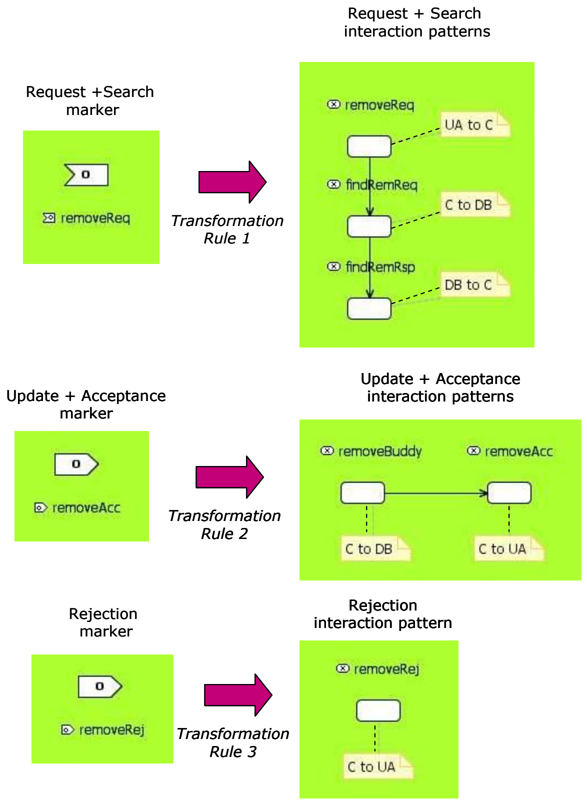


Figure 70 Remove Buddy: mappings for SStoSDRM transformation rules definition

The mappings in *Figure 70* relate SS markers to SDCM interaction patterns according to the following transformation rules:

1. Transformation rule 1 creates a mapping of an SS marker with name removeReq onto the combination of a *request* interaction pattern with name removeReq and a *search* interaction pattern with name findRemReq and findRemRsp. In order to achieve this, for each *user input* element with name removeReq that is found in the SS source model, transformation rule 1 generates in the SDRM target model the following elements: (1) a *refined data action* element of type UA to C and name removeReq, (2) a *refined data action* element of type C to DB and name findRemReq, (3) a *refined data action* element of type DB to C and name findRemRsp, (4) an *enabling relation* element between the refined data actions removeReq of type UA to C and findRemReq of type C to DB, and (5) an *enabling relation* element between the refined data actions findRemReq of type C to DB and findRemRsp of type DB to C.
2. Transformation rule 2 creates a mapping of an SS marker with name removeAcc onto the combination of an *update* interaction pattern with

name removeBuddy and an *acceptance* interaction pattern with name removeAcc. In order to achieve this, for each *user output* element with name removeAcc that is found in the SS source model, transformation rule 2 generates in the SDRM target model the following elements: (1) a *refined data action* element of type C to DB and name removeBuddy, (2) a *refined data action* element of type C to UA and name removeAcc, and (3) an *enabling relation* element between the refined data actions removeBuddy of type C to DB and removeAcc of type C to UA.

3. Transformation rule 3 creates a mapping of an SS marker with name removeRej onto a *rejection* interaction pattern with name removeRej. In order to achieve this, for each *user output* element with name removeRej that is found in the SS source model, transformation rule 3 generates a *refined data action* element of type C to UA and name removeRej in the SDRM target model.

The transformation rules mentioned above are specific to the Remove Buddy function of the Live Contacts application. In order to make these transformation rules available for reuse both in other functions of the Live Contacts application and in different applications than Live Contacts, we have generalised these rules as shown in *Table 6*. In these generalised rules, the function name used as tag for a specific rule is replaced by a more general tag. Therefore, the "remove" tag used to characterize the elements of the Remove Buddy function is replaced by an <x> tag. For example, the removeReq name is replaced by an <x>Req name in *Table 6*.

Table 6 Remove Buddy: transformation rules generalization

Transformation rule	Source element	Target element(s)
Transformation rule 1	User Input – Name: <x>Req	Refined Data Action – Name: <x>Req – Type: UA to C
		Refined Data Action – Name: find<x>Req – Type: C to DB
		Refined Data Action – Name: find<x>Rsp – Type: DB to C
		Enabling Relation – Enabling source: Refined Data Action – Name: <x>Req – Type: UA to C – Enabling target: Refined Data Action – Name: find<x>Req – Type: C to DB
		Enabling Relation – Enabling source: Refined Data Action – Name: find<x>Req – Type: C to DB – Enabling target: Refined Data Action – Name: find<x>Rsp – Type: DB to C
Transformation rule 2	User Output – Name: <x>Acc	Refined Data Action – Name: <x>Buddy – Type: C to DB
		Refined Data Action – Name: <x>Acc – Type: C to UA
		Enabling Relation – Enabling source: Refined Data Action – Name: <x>Buddy – Type: C to DB – Enabling target: Refined Data Action – Name: <x>Acc – Type: C to UA
Transformation rule 3	User Output – Name: <x>Acc	Refined Data Action – Name: <x>Rej – Type: C to UA

6.4.2 Proximity refinement transformation

Figure 71 shows the source and target models for the transformation of the Proximity service specification discussed in Section 6.2 into the Proximity service design refined model discussed in Section 6.3.

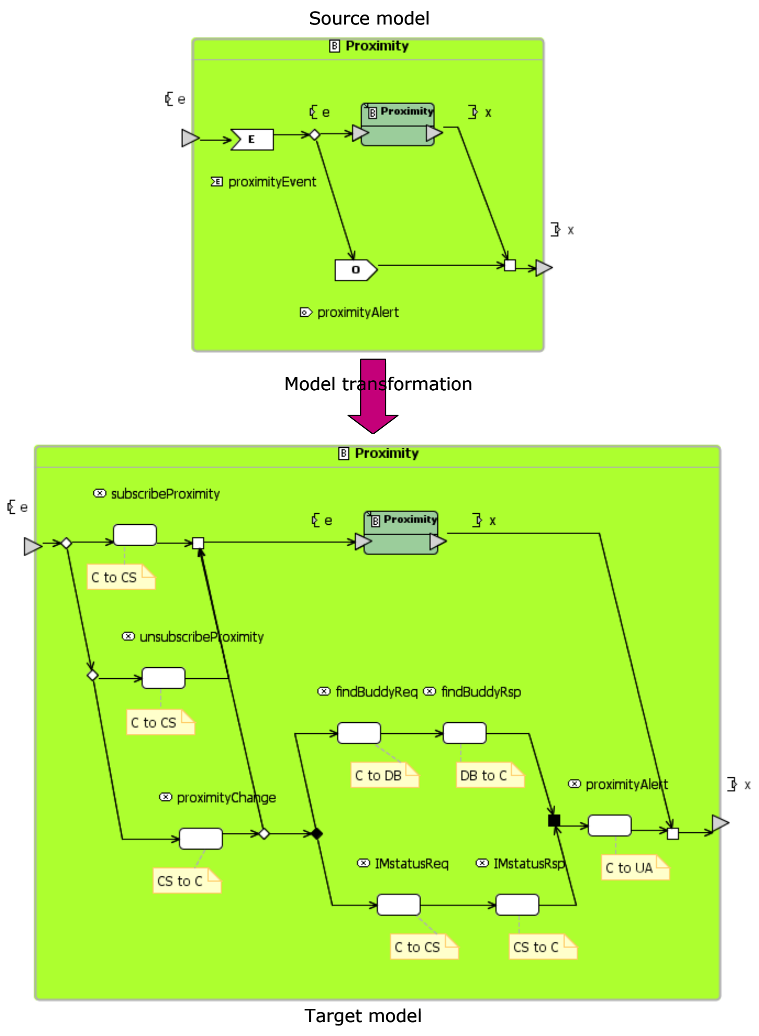


Figure 71 Proximity: source and target models for the SStoSDRM transformation

Figure 71 shows at a glance that the Proximity refinement transformation adds detail to the target model and, at the same time, preserves the behaviour structure of the source model. Analogously to the Remove Buddy refinement, also for the Proximity refinement we have defined transformation rules in order to map interaction markers in the SS source model onto interaction patterns in the SDRM target model. We have also defined transformation rules in order to map the SS behaviour structure, such as, for example, *enabling relation*, *or-split*, *or-join*, *and-split*, *and-join* elements in *Figure 71*, onto corresponding behaviour structure elements in the SDRM target model. For sake of readability, we could not present the complete set of transformation rules and *Figure 72* shows the mappings that we have used to define the transformation rules from interaction markers to interaction patterns, which we consider the most significant transformation rules for the purpose of this thesis.

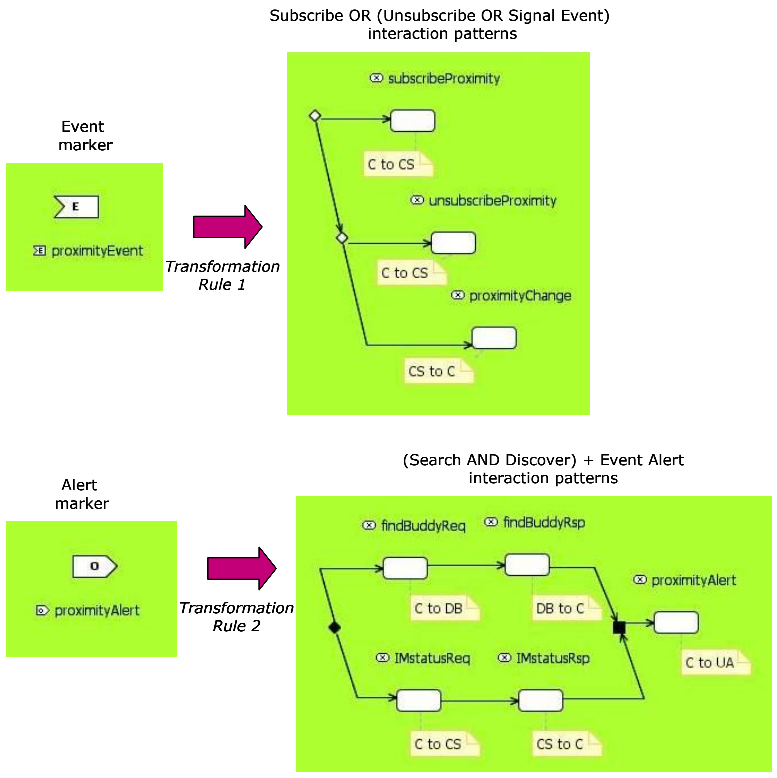


Figure 72 Proximity: mappings for SStoSDRM transformation rules definition

The mappings in *Figure 72* relate SS markers to SDCM interaction patterns according to the following transformation rules:

1. Transformation rule 1 creates a mapping of an SS marker with name proximityEvent onto the combination of a *subscribe* interaction pattern with name subscribeProximity, an *unsubscribe* interaction pattern with name unsubscribeProximity, and a *signal event* interaction pattern with name proximityChange. In order to achieve this, for each *event* element with name proximityEvent that is found in the SS source model, transformation rule 1 generates in the SDRM target model the following elements: (1) a *refined data action* element of type C to CS and name subscribeProximity, (2) a *refined data action* element of type C to CS and name unsubscribeProximity, (3) a *refined data action* element of type CS to C and name proximityChange, (4) an *enabling relation* element between an *or-split* element and the refined data action subscribeProximity of type C to CS, (5) an *enabling relation* element between two *or-split* elements, (6) an *enabling relation*

element between an *or-split* element and the refined data action unsubscribeProximity of type C to CS, and (7) an *enabling relation* element between an *or-split* element and the refined data action proximityChange of type CS to C.

2. Transformation rule 2 creates a mapping of an SS marker with name proximityAlert onto the combination of a *search* interaction pattern with name findBuddyReq and findBuddyRsp, a *context query* interaction pattern with name IMstatusReq and IMstatusRsp, and an *event alert* interaction pattern with name proximityAlert. In order to achieve this, for each *user output* element with name proximityAlert that is found in the SS source model, transformation rule 2 generates in the SDRM target model the following elements: (1) a *refined data action* element of type C to DB and name findBuddyReq, (2) a *refined data action* element of type DB to C and name findBuddRsp, (3) a *refined data action* element of type C to CS and name IMstatusReq, (4) a *refined data action* element of type CS to C and name IMstatusRsp, (5) a *refined data action* element of type C to UA and name proximityAlert, (6) an *enabling relation* element between an *and-split* element and the refined data action findBuddyReq of type C to DB, (7) an *enabling relation* element between the refined data actions findBuddyReq of type C to DB and findBuddRsp of type DB to C, (8) an *enabling relation* element between the refined data action findBuddRsp of type DB to C and an *and-join* element, (9) an *enabling relation* element between an *and-split* element and the refined data action IMstatusReq of type C to CS, (10) an *enabling relation* element between the refined data actions IMstatusReq of type C to CS and IMstatusRsp of type CS to C, (11) an *enabling relation* element between the refined data action IMstatusRsp of type CS to C and an *and-join* element, and (12) an *enabling relation* element between an *and-join* element and the refined data action proximityAlert, of type C to UA.

The transformation rules mentioned above are specific to the Proximity function of the Live Contacts application. In order to make these transformation rules available for reuse both in similar functions of the Live Contacts application and in different applications than Live Contacts, we have generalised these rules as shown in *Table 7* and *Table 8*. In these generalised rules, the function name used as tag for a specific rule is replaced by a more general tag. Therefore, the “proximity” tag used to characterize the elements of the Proximity function is replaced by an <x> tag. For example, in *Table 7*, the proximityEvent name is replaced by an <x>Event name. *Table 7* shows the generalization of transformation rule 1 described above.

Table 7 Proximity: transformation rule 1 generalization

Transformation rule	Source element	Target elements
Transformation rule 1	Event – Name: <x>Event	Refined Data Action – Name: subscribe<x> – Type: C to CS
		Refined Data Action – Name: unsubscribe<x> – Type: C to CS
		Refined Data Action – Name: <x>Change – Type: CS to C
		Enabling Relation – Enabling source: OrSplit – Enabling target: Refined Data Action – Name: subscribe<x> – Type: C to CS
		Enabling Relation – Enabling source: OrSplit – Enabling target: OrSplit
		Enabling Relation – Enabling source: OrSplit – Enabling target: Refined Data Action – Name: unsubscribe<x> – Type: C to CS
		Enabling Relation – Enabling source: OrSplit – Enabling target: Refined Data Action – Name: <x>Change – Type: CS to C

Table 8 shows the generalization of transformation rule 2 described above. The "proximity" tag used to characterize the elements of the Proximity function is replaced by an <x> tag. In contrast, the "Buddy" and "IMstatus" tags used to characterize (context) information values handled by the Proximity function are replaced by an <y> tag.

Table 8 Proximity: transformation rule 2 generalization

Transformation rule	Source element	Target elements
Transformation rule 2	User Output – Name: proximityAlert	Refined Data Action – Name: find<y>Req – Type: C to DB
		Refined Data Action – Name: find<y>Rsp – Type: DB to C
		Refined Data Action – Name: <y>Req – Type: C to CS
		Refined Data Action – Name: <y>Rsp – Type: CS to C
		Refined Data Action – Name: <x>Alert – Type: C to UA
		Enabling Relation – Enabling source: AndSplit – Enabling target: Refined Data Action – Name: find<y>Req – Type: C to DB

		Enabling Relation
		– Enabling source: Refined Data Action
		  – Name: find<y>Req
		  – Type: C to DB
		– Enabling target: Refined Data Action
		  – Name: find<y>Rsp
		  – Type: DB to C

		Enabling Relation – Enabling source: Refined Data Action – Name: find<y>Rsp – Type: DB to C – Enabling target: AndJoin
		Enabling Relation – Enabling source: AndSplit – Enabling target: Refined Data Action – Name: <y>Req – Type: C to CS

		Enabling Relation
		– Enabling source: Refined Data Action
		  – Name: <y>Req
		  – Type: C to CS
		– Enabling target: Refined Data Action
		  – Name: <y>Rsp
		  – Type: CS to C

		Enabling Relation – Enabling source: Refined Data Action – Name: <y>Rsp – Type: CS to C – Enabling target: AndJoin
		Enabling Relation – Enabling source: AndJoin – Enabling target: Refined Data Action – Name: <x>Alert – Type: C to UA

6.5 Service Design Component Model

According to our methodology, the Remove Buddy and Proximity service design refined models in *Figure 67* and *Figure 68*, respectively, and all the refinements of the other service functions should be further synthesized in a service design component model (SDCM) that consists of executable patterns. Therefore, we have realised an *SDRMtoSDCM* transformation that maps an SDRM model in A-MUSE DSL onto an SDCM model in ISDL. The SDRM model represents the interactions among components of our refrence architecture, while the SDCM model also represents the internal behaviour of these components as an orchestration from the perspective of the coordinator, which orchestrates all the other components of our reference architecture.

The SDRM to SDCM tranformation has been manually realised by assigning SDRM interaction patterns to SDCM executable patterns. *Figure 73* depicts this assignment for the Remove Buddy service design model in *Figure 67*, which represents a composite interaction pattern that consists of five basic interaction patterns (see Section 6.4.1). The Remove Buddy composite interaction pattern is an instance of the more general *user request with acceptance or rejection* composite pattern, which allows a user to make a request to the system followed by confirmation whether the required task has been successfully performed or not.

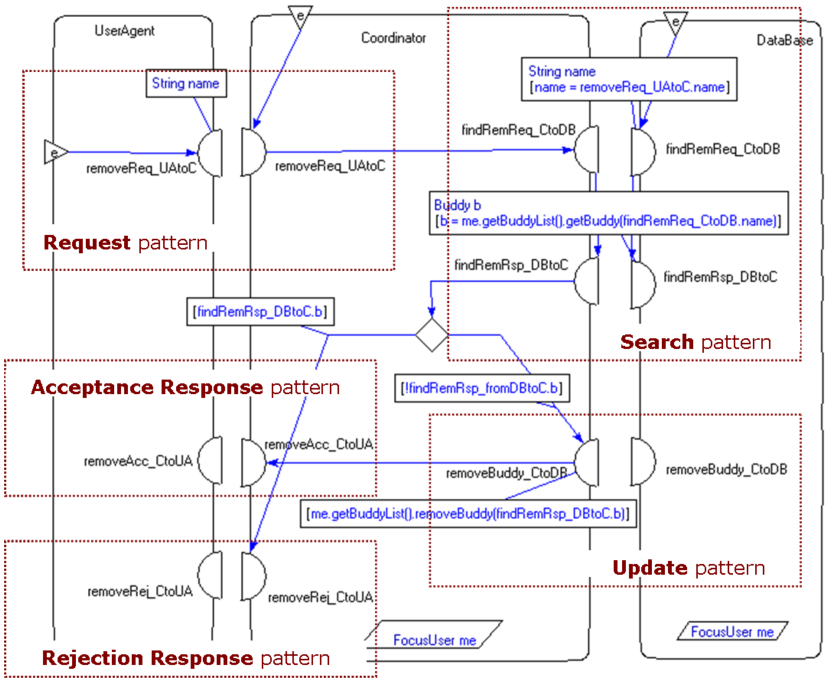


Figure 73 Mappings of SDRM interaction patterns onto SDCM executable patterns

Figure 73 depicts the behaviour of the user agent, the coordinator and the database considering their interactions. Dashed lines in *Figure 73* indicate the assignment of basic patterns to these components.

6.6 Discussion

When realising the *SStoSDRM refinement* transformation (see Section 6.4), we defined transformation rules to automate the mappings of SS interaction markers onto SDRM interaction patterns. During this work, we realised that composite patterns, such as, for example, the Remove Buddy in *Figure 67*, are not very flexible to be mapped, since they represent complex behaviours with a fixed structure. Therefore, our transformation rules became large and complex. However, since the library of composite services and patterns we have defined is rather small, the benefit of mapping composite patterns is that the number of transformation rules we needed to create was also small. In contrast, the library of basic patterns that can be used to configure composite 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 has to be considered.

When realising the *SDRMtoSDCM synthesis* transformation (see Section 6.5), we learned that the assignment of interaction patterns to executable patterns was quite straightforward. However, we noticed that some synchronization and concurrency issues of interacting components should be considered. For example, the coordinator component has to schedule somehow the execution of composite patterns represented as behaviour instances in *Figure 63*. 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, our A-MUSE and ISDL models represent only one user instance interacting with the system. In reality, the coordinator has to handle multiple user instances running at the same time. These aspects motivated us to investigate on transition systems, which are known to be suitable to handle synchronization issues. Our solution for behaviour synthesis based on transition systems is discussed in Chapter 7.

Chapter 7

Behaviour Synthesis using Transition Systems

This chapter presents a behaviour synthesis technique that uses Labelled and Modal Transition Systems (LTSs and MTSs) to model the PIM level of our methodology. This technique synthesises an executable behaviour of the application under development from the perspective of a specific component, namely the coordinator component in the case of our reference architecture. This synthesized behaviour corresponds to the *service design component model (SDCM)* of our methodology, while the behaviour used as starting point for the synthesis corresponds to our *service design refined model (SDRM)*, as discussed in Section 4.1. Therefore, the proposed technique focuses on the *SDRMtoSDCM synthesis transformation* of our methodology, which copes with behaviour models that reveal the application internal architecture. This chapter presents the source and target models of the *SDRMtoSDCM* transformation using the Live Contacts running example, and discusses the synthesis steps for realising this transformation. The source model is represented using a combination of A-MUSE DSL scenarios and Fluent Linear Temporal Logic (FLTL) safety properties. Both models are first synthesized in two LTSs, one from scenarios and one from properties, respectively. These LTSs are further synthesized in MTSs, which are finally merged into one MTS from properties and scenarios that represents the target model of our *SDRMtoSDCM synthesis transformation*.

This chapter is organised as follows: Section 7.1 gives an overview of our synthesis approach, Section 7.2 discusses the details of the synthesis from A-MUSE DSL scenarios, Section 7.3 presents the synthesis from FLTL safety properties, Section 7.4 merges the results obtained in the synthesis from scenarios and properties, and, finally, Section 7.5 discusses our experience with this technique.

7.1 Synthesis Approach

The technique proposed by Uchitel et al. [64, 67] consists of the combination of behaviour synthesis from *scenarios*, which allow to represent a limited set of *required* behaviours that the modelled application can assume, and behaviour synthesis from *properties*, which allow to represent a large set of *possible* acceptable behaviours that the modelled application can assume. Modal Transitions Systems (MTSs) can be used to capture the middle ground between scenarios and properties, since MTSs allow the distinction between *required* and *possible* behaviours (see Section 4.1). The technique in [64, 67] allows to synthesize two MTSs, one from scenarios and one from properties, and merge them in a resulting MTS that is demonstrated to preserve the original properties and scenarios. *Figure 74* shows the approach for behaviour synthesis from properties and scenarios that we have used in this thesis according to [64].

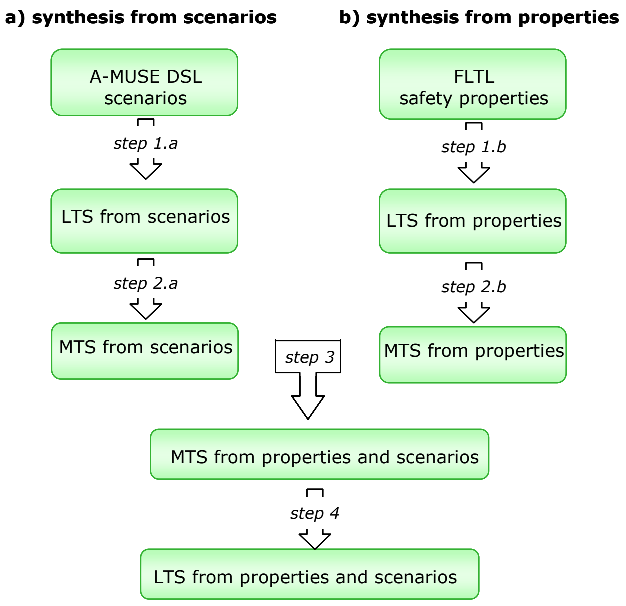


Figure 74 Synthesis approach using A-MUSE DSL as scenario modelling language

Although Message Sequence Charts (MSCs) are used in [64] to represent scenarios, none specific technique or language for scenarios representation is prescribed. Therefore, we used the A-MUSE DSL to represent our scenarios, since it allowed us to represent (1) sequences of interaction among architecture components, similarly to basic MSCs used in [67], (2) the control flow between these sequences of interactions, similarly to high-level MSCs used in [67].

Our synthesis approach starts by specifying an A-MUSE DSL scenario, which represents the service design refined model (SDRM) that reveals the components of our architecture. From this scenario we synthesized an LTS that represents the *required* behaviour of the system from the perspective of the coordinator component, which orchestrates all the other components of our reference architecture (step 1.a in *Figure 74*). From the resulting LTS, we synthesized an MTS that considers also *possible* (but not necessarily required) behaviour in order to extend the limited set of example behaviours represented in the LTS (step 2.a in *Figure 74*). In parallel, we realised a behaviour synthesis from FLTL properties that extend the considered A-MUSE DSL scenario (steps 1.b and 2.b in *Figure 74*). This synthesis from properties resulted in an MTS that specifies *possible* behaviour that does not violate the desired properties. Finally, we merged the two MTSs from properties and scenarios in one MTS (step 3 in *Figure 74*), which represents the behaviour of the system from the perspective of our coordinator and corresponds to the service design component model (SDCM) of our methodology. Therefore, the approach in *Figure 74* performs the synthesis transformation from SDRM to SDCM. The final MTS representing our SDCM model should be taken as input to create an implementation of the coordinator using some specific technology. However, since this MTS represents both *required* and *possible* behaviour, it can be further refined in an LTS that represents only *required* behaviour (step 4 in *Figure 74*) by recursively applying the synthesis approach for new properties and scenarios.

7.2 Synthesis from Scenario

Figure 75 to *Figure 77* show the A-MUSE DSL scenario we have used as starting point of our synthesis approach. This scenario represents the Remove Buddy and Proximity functions of the Live Contacts application already adopted as running example in Chapter 6.

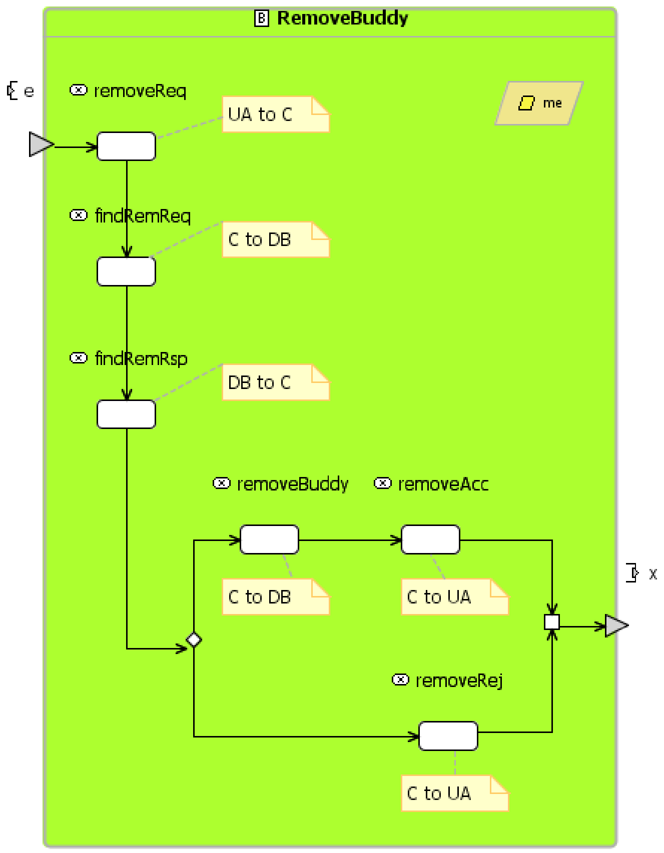


Figure 75 Scenario in A-MUSE DSL, part 1

Figure 75 shows the Remove Buddy function at the SDRM level (see Section 6.3). Since the SDRM level represents a behaviour distributed among system components, *Figure 75* shows with annotations the components involved in the interactions, which are the user agent (UA), coordinator (C) and database (DB) components of our reference architecture. A removeReq interaction between the user agent and coordinator (UA to C) is followed by a findRemReq request (C to DB) and a findRemRsp response (DB to C) to/from the database. Depending on the database response, the coordinator removes the buddy from the users' list (removeBuddy) and sends a positive response to the user agent (removeAcc). Otherwise, a negative response removeRej is sent to the user agent.

Figure 76 Scenario in A-MUSE DSL, part 2

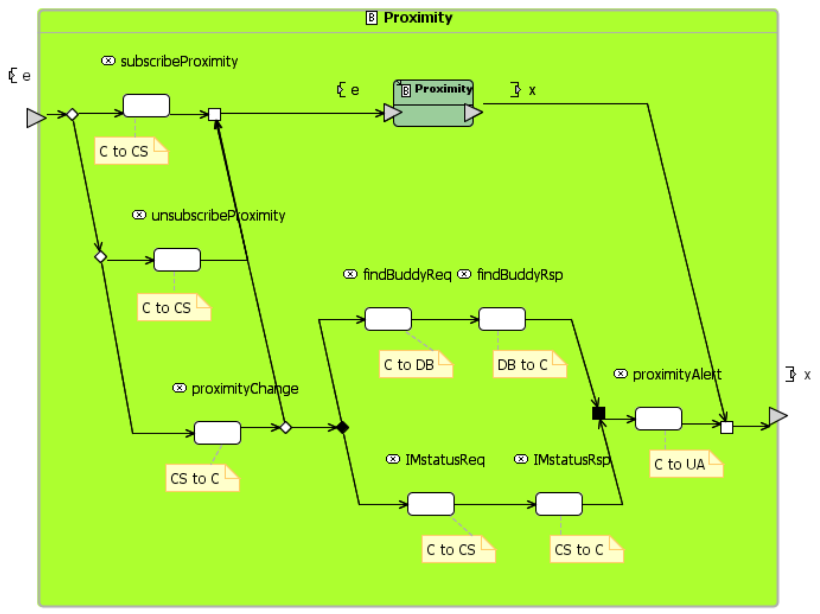


Figure 76 shows the Proximity function at the SDRM level (see Section 6.3), which involves the coordinator (C), context source (CS), database (DB) and user agent (UA) components of our reference architecture. *Figure 76* shows that the coordinator can subscribe for a proximity event to the context sources (C to CS) via a subscribeProximity request. Upon the occurrence of a proximity event, the context source notifies the coordinator with a proximityChange interaction (CS to C). After a request/response interaction with the database (findBuddyReq and findBuddyRsp) and the context sources (IMstatusReq and IMstatusRsp), the coordinator generates a proximityAlert to the user agent (C to UA) in order to notify the user about the proximity event. Any time after subscribing, the coordinator can also unsubscribe to the proximity event with the unsubscribeProximity request in *Figure 76*.

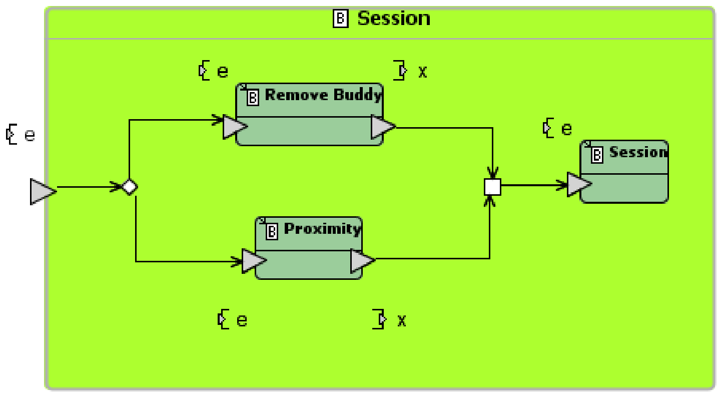


Figure 77 Scenario in A-MUSE DSL, part 3

Figure 77 shows the (Live Contacts) Session behaviour, which represents the way our application handles the Remove Buddy and Proximity functions mentioned above. These functions are two alternative threads of control. When a chosen thread ends, a Session instance follows and a new alternative can be chosen again.

7.2.1 From A-MUSE DSL scenarios to LTSs

According to the technique for synthesis from scenarios in [64], we have used the A-MUSE DSL scenario described above to synthesize an LTS. In order to achieve this, we have adapted the steps in [67] as follows:

a. For each component involved in the Remove Buddy function, namely the user agent, coordinator and database components, we synthesized an LTS using the scenario in *Figure 75*. By doing so, we obtained three *partial* LTSs that describe part of the scenario from the perspective of a specific component.
b. For each component involved in the Proximity function, namely the context source, coordinator, database and user agent components, we synthesized an LTS using the scenario in *Figure 76*. By doing so, we obtained four *partial* LTSs that describe part of the scenario from the perspective of a specific component.
c. For each component involved in the whole scenario, namely the context source, coordinator, database and user agent components, we combined the LTS obtained for the Remove Buddy function (see *step a* above) with the LTS obtained for the Proximity function (see *step b* above) using the scenario in *Figure 77*. By doing so, we obtained four *complete* LTSs that describe the whole scenario from the perspective of a specific component.
d. We combined the partial LTSs obtained in *step c* in one LTS that describes the whole scenario from the perspective of the system.

In the following we elaborate on these steps and provide examples. The following definition of LTS provides some background to understand our examples. More details on LTSs can be found in [64].

Definition 11 Labelled Transition Systems

Let States *be a universal set of states, and* Act *be a universal set of observable action labels. An LTS is a tuple* (S, A, Δ, s_0)*, where* S ⊆ States *is a finite set of states,* A ⊆ Act *is a set of labels,* Δ ⊆ (S × A × S) *is a transition relation, and* s_0 ∈ S *is the initial state.*

In our scenario, the finite set of action labels consists of A = {removeReq, findRemReq, findRemRsp, removeBuddy, removeAcc, removeRej, subscribeProximity, unsubscribeProximity, proximityChange, findBuddyReq, findBuddyRsp, IMstatusReq, IMstatusRsp, proximityAlert}. These labels represent the interactions shown in *Figure 75* and *Figure 76*. An LTS that represents the behaviour of a specific component uses (part of) these action labels. For example, *Figure 78* shows the LTSs that represent the behaviour of the user agent component.

Figure 78 User agent behaviour

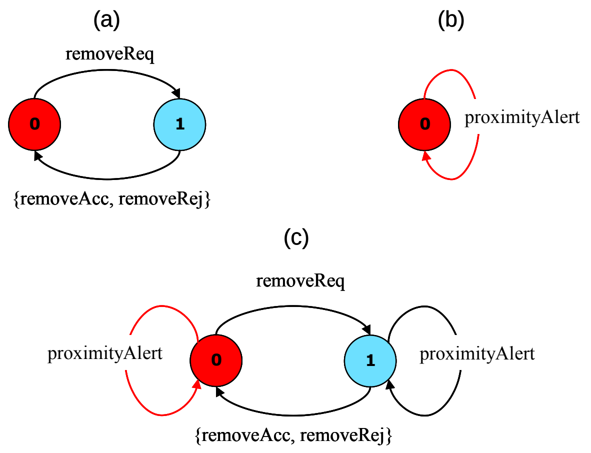


Figure 78.(a) shows an LTS that represents the user agent behaviour in the Remove Buddy function (see *step a* mentioned above). This LTS consists of the states S = {so, s1} and action labels A = {removeReq, removeBuddy, removeAcc}, which are the interactions that involve the user agent (UA) in the scenario in *Figure 75*. In this LTS, the user agent sends a request (removeReq) and waits for a confirmation whether this request has been accomplished (remove Acc) or not (remove Rej).

Figure 78.(b) shows an LTS that represents the user agent behaviour in the Proximity function (see *step b* mentioned above). This LTS consists of only one state S = {so} and action label A = {proximityAlert}, which is the only interaction that involve the user agent (UA) in the scenario in *Figure 76*. This

LTS shows that the user agent is in a state that waits for a proximityAlert to happen.

Figure 78.(c) shows an LTS that represents the combined behaviour of the user agent in the Remove Buddy and Proximity functions (see *step c* mentioned above). This LTS consists of the states S = {so, s1} and action labels A = {removeReq, removeBuddy, removeAcc, proximityAlert}.

Figure 79 shows the LTSs that represent the behaviour of the coordinator component.

Figure 79 Coordinator behaviour

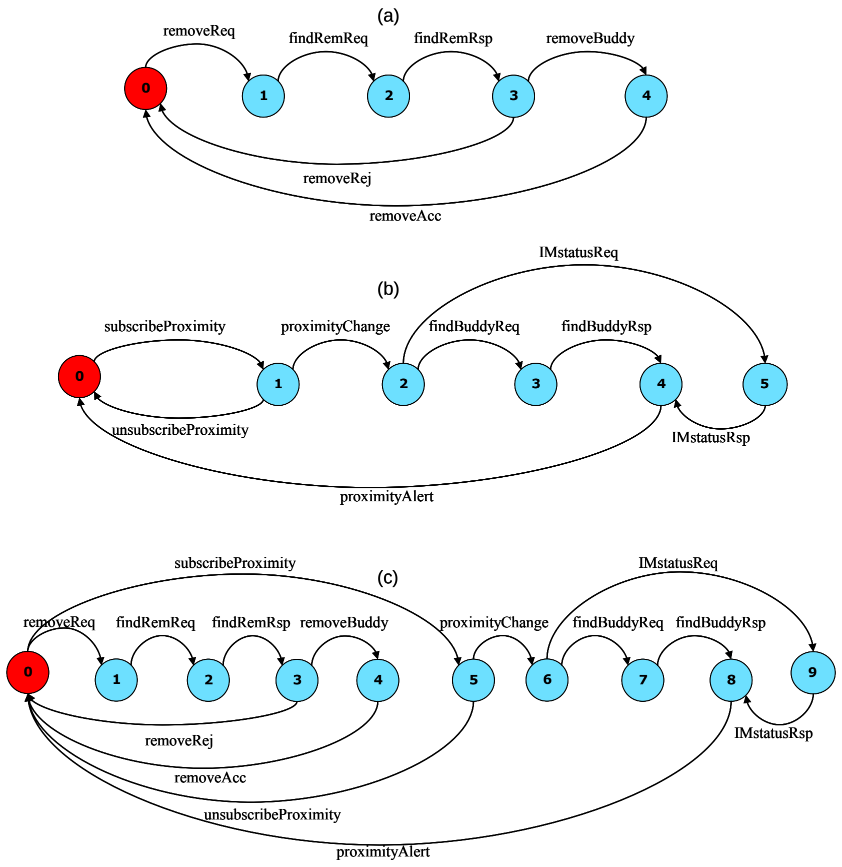


Figure 79.(a) represents the LTS synthesized from the Remove Buddy scenario in *Figure 75*. This LTS consists of the states S = {so, s1, s2, s3, s4} and action labels A = {removeReq, findRemReq, findRemRsp, removeBuddy, removeAcc}, which are the interactions that involve the coordinator (C), namely all the interactions represented in *Figure 75*.

Figure 79.(b) shows the coordinator behaviour in the Proximity function (see *step b* mentioned above). This LTS consists of the states S = {so, s1, s2, s3, s4} and action labels A = {subscribeProximity, unsubscribeProximity, proximityChange, findBuddyReq, findBuddyRsp, IMstatusReq, IMstatusRsp, proximityAlert}, which are obtained from the interactions that involve the coordinator (C), namely all the interactions in *Figure 76*. The A-MUSE DSL scenario in *Figure 76* shows that the interactions {findBuddyReq, findBuddyRsp} and {IMstatusReq, IMstatusRsp} are executed concurrently by the coordinator. We represented this in our LTS with the transitions (s2, findBuddyReq, s3), (s3, findBuddyRsp, s4), (s2, IMstatusReq, s5), and (s4, IMstatusRsp, s5).

Figure 79.(c) shows the combined LTS for the Remove Buddy and Proximity functions (see *step c* mentioned above). This LTS has been combined using the scenario in *Figure 77*, which represents the Remove Buddy and Proximity functions as alternative choices. Therefore, the corresponding LTS in *Figure 79* also represents these functions as alternative choices.

Figure 80 shows the LTSs that represent the behaviour of the database component according to steps a, b and c mentioned above.

Figure 80 Database behaviour

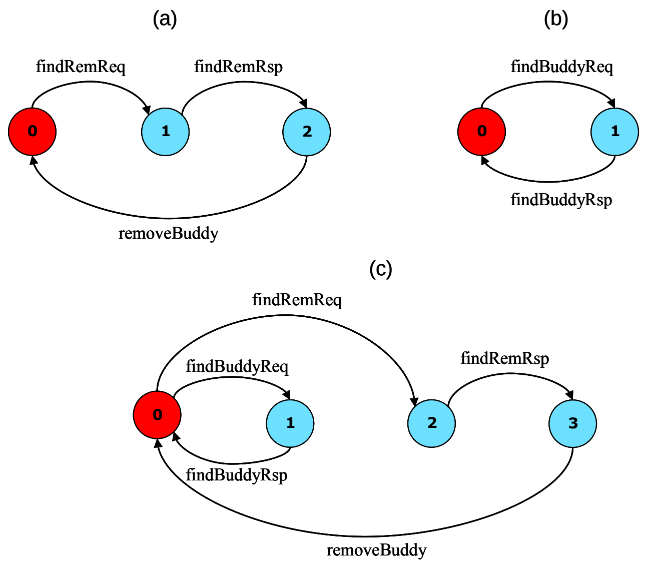


Figure 81 shows the LTS that represents the behaviour of the context source component according to *steps b* mentioned above. Since the context source component is only involved in the Proximity function, steps a and c discussed above were not necessary for this component.

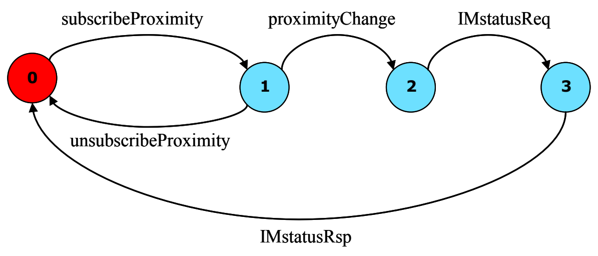


Figure 81 Context Source behaviour

Figure 82 shows the final LTS for our scenario. This LTS has been synthesized following step d mentioned above, namely by combining the partial LTSs represented from *Figure 78* to *Figure 81* (*part c*). Since the coordinator component orchestrates the other components of our reference architecture and is involved in all the interactions considered in our scenario, our final LTS is synthesized from the perspective of the coordinator. Therefore, *Figure 82* corresponds to the LTS already presented in *Figure 79*, *part c*.

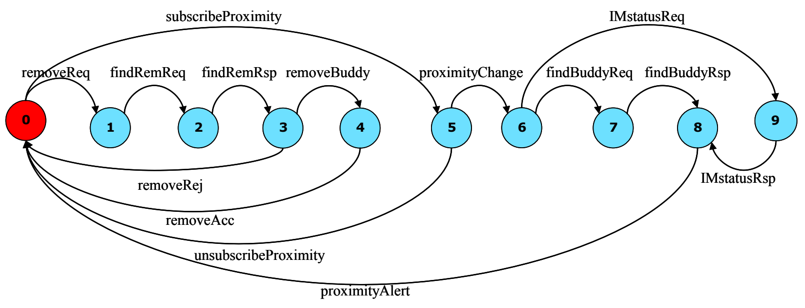


Figure 82 LTS from A-MUSE DSL scenario

In the LTS in *Figure 82*, the finite set of states consists of S = {s1, s2, s3, s4, s5, s6, s7, s8, s9} and the set of action labels consists of A = {removeReq, findRemReq, findRemRsp, removeBuddy, removeAcc, removeRej, subscribeProximity, unsubscribeProximity, proximityChange, findBuddyReq, findBuddyRsp, IMstatusReq, IMstatusRsp, proximityAlert}.

7.2.2 From LTSs to MTSs

Definition 12 Modal Transition Systems

An MTS is a structure (S, A, Δ^r, Δ^p, s_0)*, where* $\Delta^r \subseteq \Delta^p$*,* (S, A, Δ^r, s_0) *is an LTS representing* required *transitions of the system and* (S, A, Δ^p, s_0) *is an LTS representing* possible *(but not necessarily required) transitions.*

The LTS in *Figure 82* represents only *required* transitions that we explicitly decided to include in our scenario. However, we should also add to this scenario *possible* (but not necessarily required) transitions that do not violate the system behaviour. As discussed in Section 4.1, a behaviour model synthesized from scenarios includes a limited set of example behaviours that the modelled application can assume. However, there are other possible behaviours that have not been considered yet. Therefore, we synthesized the MTS in *Figure 83* that considers also this *possible* behaviour. In this MTS, *possible* transitions are distinguished from *required* transitions by a question mark that follows the transition label. In order to avoid clogging *Figure 83*, only part of our MTS is shown, which corresponds to states S = {s0, s1, s2, s3, s4, s5} in *Figure 82*. State s6 in *Figure 83* is a sink state that we have added during the synthesis according to the algorithm in [64], which is explained in the sequel. This sink state s6 should not be confused with the state s6 in *Figure 82*. The set of action labels considered in *Figure 83* is A = {removeReq, findRemReq, findRemRsp, removeBuddy, removeAcc, removeRej, subscribeProximity, unsubscribeProximity}. Some of these action labels are shown in *Figure 83* in a short form for illustration purposes. For example, the remReq label is used instead of removeReq, without loss of clarity.

We have obtained the MTS in *Figure 83* from the LTS in *Figure 82* using the *MTSscen algorithm* proposed in [64] as follows:

- we added a sink state to the LTS in *Figure 82*. The sink state is represented in *Figure 83* using a state s6;
- we added a *possible* looping transition (labelled with a question mark) to this sink state for every action a ∈ A. This is represented in *Figure 83* using the looping transition in state s6 with labels {remReq?, findRemReq?, findRemRsp?, remBuddy?, remAcc?, remRej?, subProximity?, unsubProximity?};
- for each state s ∈ S, such that there is no outgoing *required* transition, we added a *possible* transition to the sink state s6. For example, consider the state s0 in the LTS in *Figure 82*. The only outgoing *required* transition is (s0, remReq, s1). Therefore, in the corresponding MTS we added a *possible* transition to the sink state s6 for each action label a ∈ A, except for the remReq action label. This is represented in *Figure 83* by the transition (s0, {findRemReq?, findRemRsp?, remBuddy?, remAcc?, remRej?, subProximity?, unsubProximity?}, s6). Analogously, we have added these possible transitions to state s6 but with different labels for the remaining states s1, s2, s3, s4, s5, as shown in *Figure 83*.

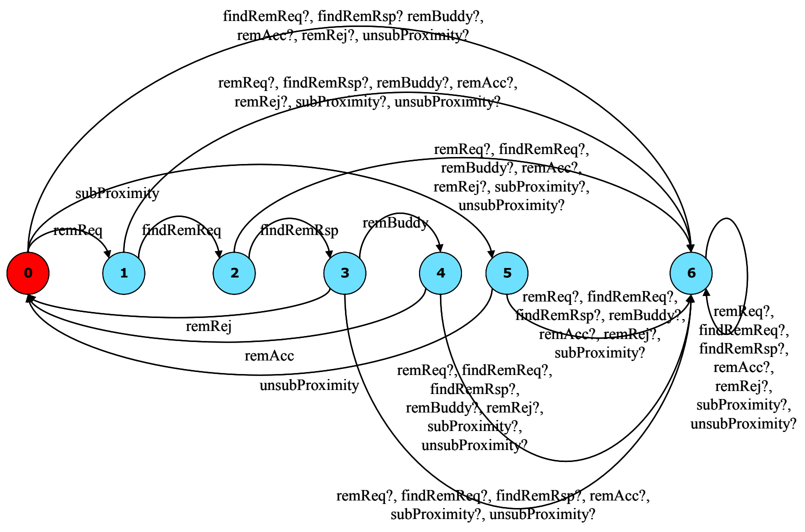

findRemReq?, findRemRsp? remBuddy?, remAcc?, remRej?, unsubProximity?
remReq?, findRemRsp?, remBuddy?, remAcc?, remRej?, subProximity?, unsubProximity?
remReq?, findRemReq?, remBuddy?, remAcc?, remRej?, subProximity?, unsubProximity?
subProximity
remReq
findRemReq
findRemRsp
remBuddy
0
1
2
3
4
5
6
remReq?, findRemReq?, findRemRsp?, remBuddy?, remAcc?, remRej?, subProximity?
remReq?, findRemReq?, findRemRsp?, remAcc?, remRej?, subProximity?, unsubProximity?
remRej
remAcc
unsubProximity
remReq?, findRemReq?, findRemRsp?, remBuddy?, remRej?, subProximity?, unsubProximity?
remReq?, findRemReq?, findRemRsp?, remAcc?, subProximity?, unsubProximity?

Figure 83 MTS from A-MUSE DSL scenario

7.3 Synthesis from Properties

The scenario discussed in Section 7.2 is not sufficient to cover all the possible interactions related to the *remove buddy* and *proximity* functions. For example, consider a user that has removed a buddy from his contact list. The removal of this buddy implies that the user should stop receiving proximity alerts about the removed buddy. Therefore, after the remBuddy action in *Figure 75*, the coordinator should unsubscribe the proximity event for that buddy with the unsubProximity action in *Figure 76*. However, this relation is not explicitly represented by the A-MUSE DSL in our models, since the two behaviours for the *remove buddy* and *proximity* functions are designed to be independent (see *Figure 77*). Therefore, we can represent this relation as a safety property, which specifyies that "nothing bad can happen". Safety properties can be expressed, for example, by using Fluent Temporal Logic (FLTL) as recommended in [64-65] because FLTL provides a uniform framework for specifying and model-checking state-based temporal properties in event-based models. The FLTL property for the interaction between remBuddy and unsubProximity is the following:

$$P = \mathbf{G}\ (\text{remBuddy} \Rightarrow \mathbf{X}\ \text{unsubProximity})$$

This formalization of P states that when the remBuddy action occurs, then the next event (**X**) to occur is the unsubProximity action. According to the technique for synthesis from properties in [64], we have generated an LTS from the safety property P, as shown in *Figure 120*. The set of action labels considered in *Figure 120* is the complete set of interactions of our *remove buddy* and *proximity* examples, namely A = {remReq, findRemReq, findRemRsp, remBuddy, remAcc, remRej, subProximity, unsubProximity, proximityChange, findBuddyReq, findBuddyRsp, IMstatusReq, IMstatusRsp, proximityAlert}.

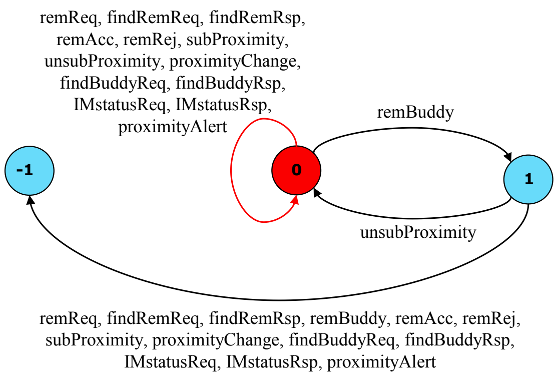

remReq, findRemReq, findRemRsp, remAcc, remRej, subProximity, unsubProximity, proximityChange, findBuddyReq, findBuddyRsp, IMstatusReq, IMstatusRsp, proximityAlert
remBuddy
-1
0
1
unsubProximity
remReq, findRemReq, findRemRsp, remBuddy, remAcc, remRej, subProximity, proximityChange, findBuddyReq, findBuddyRsp, IMstatusReq, IMstatusRsp, proximityAlert

Figure 84 LTS from properties

The LTS in *Figure 120* is a Büchi automaton *B(¬P)* [64-65] with an error state s-1 that represents all the transitions with action labels a ∈ A that violate the property P. In other words, the sequence {(s0, remBuddy, s1), (s1, unsubProximity, s0)} in *Figure 120* represent our safety property P = **G** (remBuddy ⇒ **X** (unsubProximity)), while (s1, {remReq, findRemReq, findRemRsp, remAcc, remRej, subProximity, proximityChange, findBuddyReq, findBuddyRsp, IMstatusReq, IMstatusRsp, proximityAlert}, s-1) represents the behaviour that violates the property P. For example, the sequence {(s0, remBuddy, s1), (s1, remReq, s-1)} represents undesired behaviour because it leads to the the error state s-1, while the only desired behaviour that should follow the (s0, remBuddy, s1) transition consists of (s1, unsubProximity, s0) as stated in our safety property P.

Analogously to the synthesis from scenario, in the synthesis from properties one can be interested in adding *possible* but not necessarily *required* transitions to an LTS. Therefore, we synthesized the MTS in *Figure 85* from the LTS in *Figure 120* using the *MTSprop* algorithm proposed in [64].

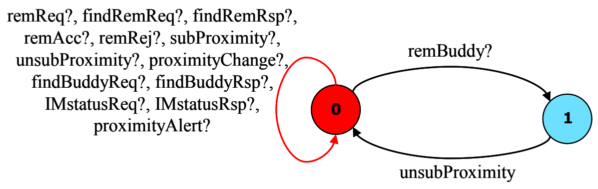

remReq?, findRemReq?, findRemRsp?,
remAcc?, remRej?, subProximity?,
unsubProximity?, proximityChange?,
findBuddyReq?, findBuddyRsp?,
IMstatusReq?, IMstatusRsp?,
proximityAlert?
remBuddy?
0
1
unsubProximity

Figure 85 MTS from properties

The MTS in *Figure 85* has been obtained by (1) removing the error state `s-1` from the LTS *Figure 120* and (2) for each state `s` ∊ `S` with more than one outgoing transition (state `s0` in our example) converting all these outgoing transitions to *possible* transitions. More details on how to generate LTSs and MTSs from properties can be found in [64].

7.4 Synthesis from Properties and Scenarios

The MTS synthesized from properties in *Figure 85* and the MTS synthesized from scenarios in *Figure 83* can be finally merged in one MTS, as shown in *Figure 86*. A merged MTS preserves the original properties and scenario intended for the system under development, as demonstrated in [64]. In this way, we can guarantee correctness and consistency of system behaviour during the development steps towards the final system implementation when using the discussed technique for behaviour synthesis from properties and scenarios.

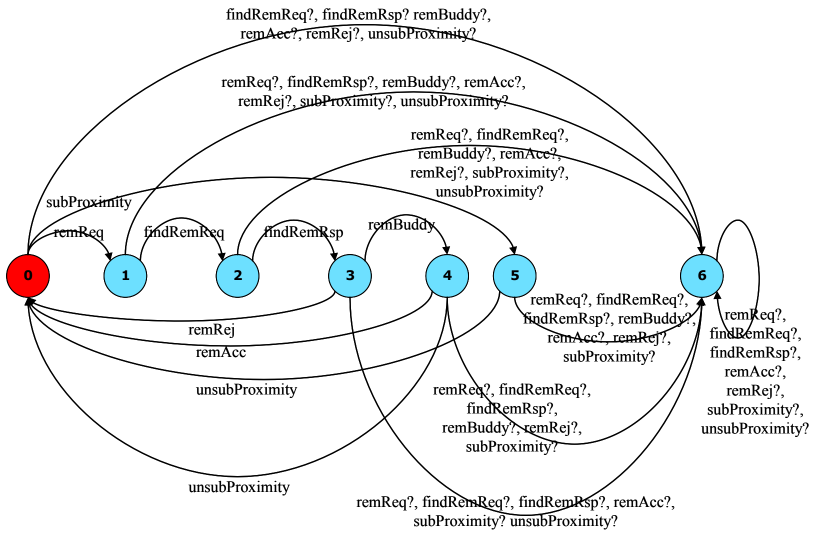


Figure 86 MTS from properties and scenarios

The MTS from properties and scenarios in *Figure 86* is rather similar to the MTS from scenarios in *Figure 83*. The differences between these two MTSs are the addition to the MTS in *Figure 86* of the *required* transition (s4, unsubProximity, s0) and the removal of the *possible* transition (s4, unsubProximity?, s6) from the MTS in *Figure 83*. In our example we have used only the property P = **G**(remBuddy ⇒ **X** unsubProximity), otherwise the example would have become too big for illustration purposes. However, additional properties can be defined to explicitly represent other relations between the remove buddy and proximity functions, as well as between other functions of the Live Contacts running example. By adding new properties, we may reduce the *possible* transitions in *Figure 86* until only *required* transitions remain. In this case, we obtain again an LTS. In practice, it may not be necessary to refine the MTS into a single LTS, since the designer may explicitly decide to leave some behavioural choices open further down in the development process [64].

7.5 Discussion

While synthesizing our coordinator component from properties and scenarios, we could extract our scenarios from the A-MUSE DSL in a straightforward way. From the A-MUSE DSL behaviour models that represent application functions, such as the *remove buddy* and *proximity* functions in our example (see *Figure 75* and *Figure 76*), we could extract sequences of interactions among architectural components, similarly to basic Message Sequence Charts used to extract secanarios in [67]. From the high-level structure of these A-MUSE DSL behaviour models (see *Figure 77*), we could extract the control flow between different functions, similarly to high-level Message Sequence Charts used to extract secanarios in [67] (see Section 4.1). Moreover, the A-MUSE DSL allowed us to raise the abstraction level of the behaviour synthesis technique in [64], which starts the synthesis from a behaviour that already reveals the architecture of the system under development. In contrast, by using the A-MUSE DSL one can specify a high-level behaviour that is independent of any specific architecture component, and exploit our transformation based on patterns (see Section 6.4) to automatically assign this behaviour to specific components. Concerning the suitability of the adopted languages, we also realised that LTSs and MTSs are an excellent means to handle concurrency and synchronization issues that arised in our methodology because of the use of patterns. However, we learned that scalability aspects are critical when using these LTSs and MTSs, since already in a limited example, such as the *remove buddy* and *proximity* functions presented in this chapter, our models became big and difficult to handle for illustration purposes.

We have created the LTSs and MTSs discussed in this chapter manually in order to assess the applicability of the behaviour synthesis technique from properties and scenarios in [64] to our methodology. The assessment was positive and we have been able to generate a merged MTS from properties and scenario that represents the behaviour of our coordinator component by systematically applying the steps prescribed in [64]. The resulting MTS corresponds to the behaviour of the coordinator component at the service design component model (SDCM) level of our methodology, while the given scenario corresponds to the service design refined model (SDRM). Therefore, the proposed technique is suitable to perform the *SDRMtoSDCM behaviour synthesis* transformation of our methodology. We have learned that this transformation can be automated using the LTSA [132] and MTSA [133] tools. We used the LTSA tool to specify the LTSs and to compose them for the synthesis purpose. We could use the MTSA tool to automatically generate the MTSs from scenarios expressed as basic Message Sequence Charts and high-level Message Sequence Charts. Finally, for scenarios represented using the A-MUSE DSL, the Medini QVT tool can be used to automatically generate these scenarios from A-MUSE DSL abstract specifications, as discussed in Chapter 6.

Chapter 8

Behaviour Refinement and Synthesis using BPMN

This chapter presents a technique that uses BPMN as modelling language for behaviour refinement and synthesis at the PIM level of our methodology. The chapter discusses the source and target models of our PIM behaviour refinement and synthesis transformations, and presents these transformations as well. The *service specification (SS)* and *service design refined model (SDRM)* are the source and target models of the *SStoSDRM behaviour refinement* transformation, respectively. The *service design refined model (SDRM)* and the *service design component model (SDCM)* are the source and target models of the *SDRMtoSDCM behaviour synthesis* transformation, respectively. The SS, SDRM and SDCM models are represented using BPMN. We realised a prototype of the *SStoSDRM refinement* and *SDRMtoSDCM synthesis* transformations using the ATL transformation engine. This chapter introduces and justifies the mappings we have used to create the ATL transformation specification taken as input in our experiment, abstracting from the specific ATL language constructs.

This chapter is organised as follows: Sections 8.1 and 8.2 present the SS and SDRM models, respectively, Section 8.3 discusses the *SStoSDRM refinement* transformation, Section 8.4 presents the SDCM model, Section 8.5 discusses the *SDRMtoSDCM synthesis* transformation and, finally, Section 8.6 discusses our experience with BPMN as modelling language.

8.1 Service Specification

The service specification represents the interactions between the system under development, considered as a black box, and its user. In the following Sections we show how these interactions can be represented in

BPMN using the *remove buddy* and *proximity* functions of the Live Contacts running example introduced in Chapter 6.

8.1.1 High-level structure

Figure 87 shows the high-level structure of the service specification for the Live Contacts running example using BPMN. This specification represents a behaviour equivalent[1] to the one expressed in the A-MUSE DSL specification shown in *Figure 62*.

Figure 87 Live Contacts service specification, SS

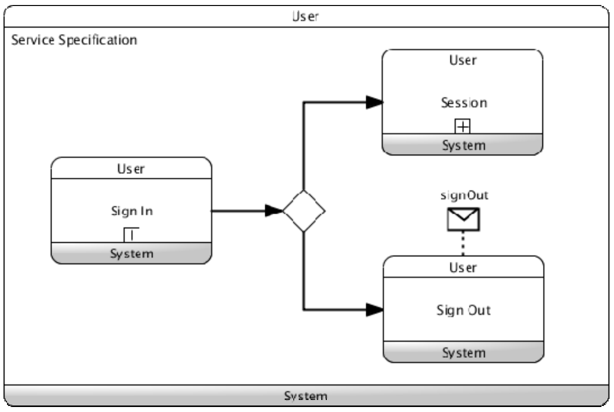


Figure 87 consists of a BPMN choreography diagram that represents the interactions between the two participants System and User, which represent the Live Contacts application and one of its users, respectively. In order to access the services offered by the system, the user must first Sign In. Once the user has signed in, he can decide to start a Session and eventually to Sign Out and exit the application. Sign In and Session are represented as a BPMN collapsed choreography sub-process (a choreography with a "+" marker), which is a compound choreography that can be refined into a finer level of detail. In contrast, Sign Out is represented as an atomic choreography task with a signOut message flow attached to the (unshaded) participant User, which is the initiator of the choreography task. The Session choreography sub-process is shown in *Figure 88* and represents a behaviour equivalent to the one expressed in the A-MUSE DSL session specification shown in *Figure 63*.

[1] Our notion of equivalence is intuitive and no formal verification of this notion was performed in this thesis.

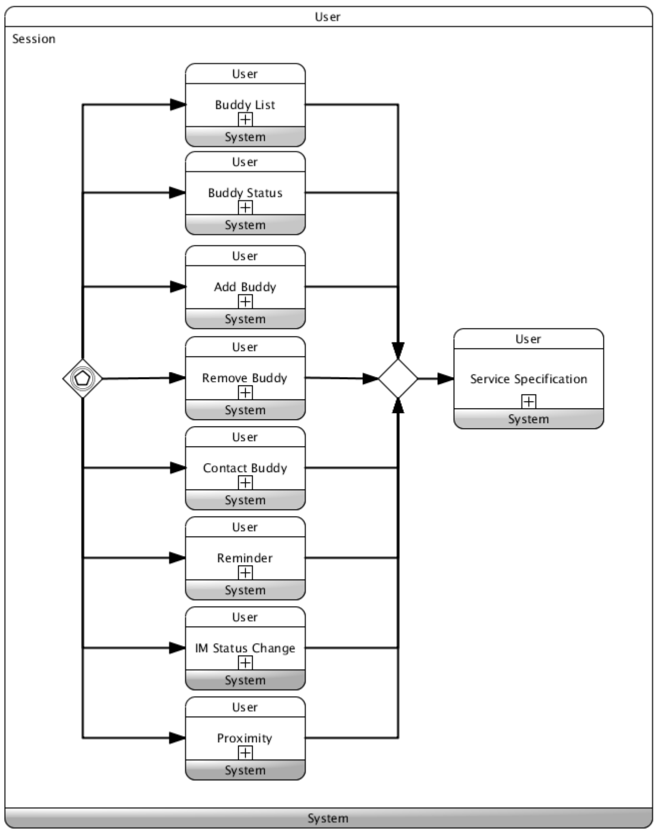


Figure 88 SS, Session

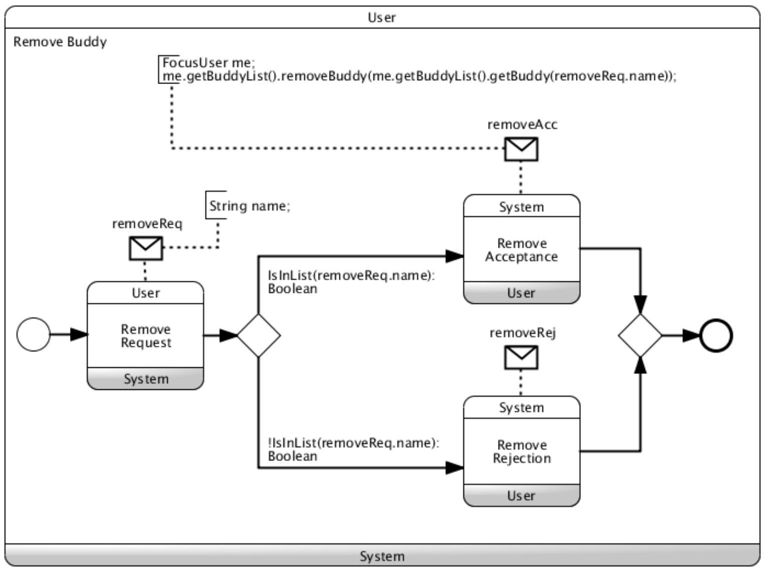
Once the user has entered a session, the application waits for an event to happen, which is represented in *Figure 88* using an *event-based* gateway element. This event allows the user to choose between the (alternative) functions offered by the Live Contacts application and discussed in Chapter 6 (see *Table 5*). When the chosen function is completely perfomed, a new Service Specification choreography sub-process is started and a new service function can be chosen. Each service function is represented as a collapsed choreography sub-process that can be refined into further detail. In the following Sections we discuss the Remove Buddy and Proximity service functions.

8.1.2 Service functions

Figure 89 and *Figure 90* zoom into the detail of the Remove Buddy and Proximity service functions, respectively.

The Remove Buddy function (*choreography sub-process* element) in *Figure 89* consists of a *choreography task* element named Remove Request followed either by a *choreography task* element named Remove Acceptance, or a *choreography task* element named Remove Rejection. We used *choreography task* elements as *interaction markers* (see Section 3.2), namely as placeholders for abstract interactions at the SS level that correspond to (more concrete) refined interactions among specific components at the SDRM level.

Figure 89 SS, Remove Buddy



The Remove Buddy function in *Figure 89* starts with a Remove Request task in which the User initiates the interaction by sending a removeReq message to the System with the name of the buddy to be removed (String name). As a consequence, the System evaluates with an *exclusive decision gateway* element whether the required buddy is actually in the list of the user. In case this buddy is in the list, the exclusive decision is followed by a Remove Acceptance task, in which the System removes the buddy from the user list and acknowledges the user about this removal (removeAcc message). In case the buddy is not the list and cannot be removed, the exclusive decision is followed by a Remove Rejection task with a removeRej message to the user. The status information handled by the Remove Buddy function in *Figure 89* is defined in the Live Contacts information model (see Section 6.1.1). This information is shown in *Figure 89* using textual annotations attached to *message flow* elements.

The Proximity function (*choreography sub-process* element) in *Figure 90* consists of a *signal event* element named proximityEvent followed either by a *choreography task* element named Proximity Event Alert, or the termination of the Proximity function. We use these *signal event* and *choreography task* elements as *interaction markers* for the Proximity function.

Figure 90 SS, Proximity

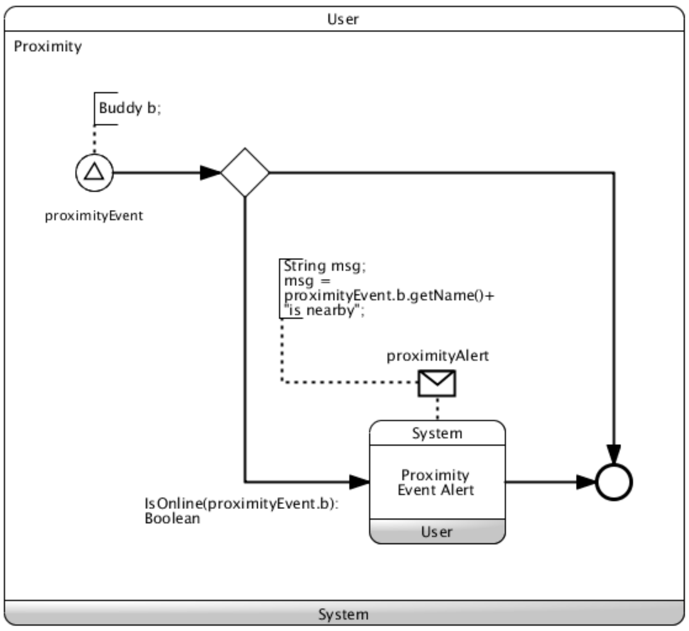


Figure 90 shows that the Proximity function starts with a proximityEvent signal. This signal notifies the occurrence of a proximity situation (see Section 6.1.2) when a buddy (Buddy b), whose IM status is “online” (IsOnline(proximityEvent.b): Boolean condition) is nearby the user. As a consequence, the application warns the user with an alert (proximityAlert message). Alternatively, namely if the IM status of the buddy is not “online” in the application, the Proximity function ends without warning the user and a new function can be chosen.

8.2 Service Design Refined Model

The service design refined model refines the functions at the SS level into more concrete interactions that are performed by components of the reference architecture, which is recalled in *Figure 91*.

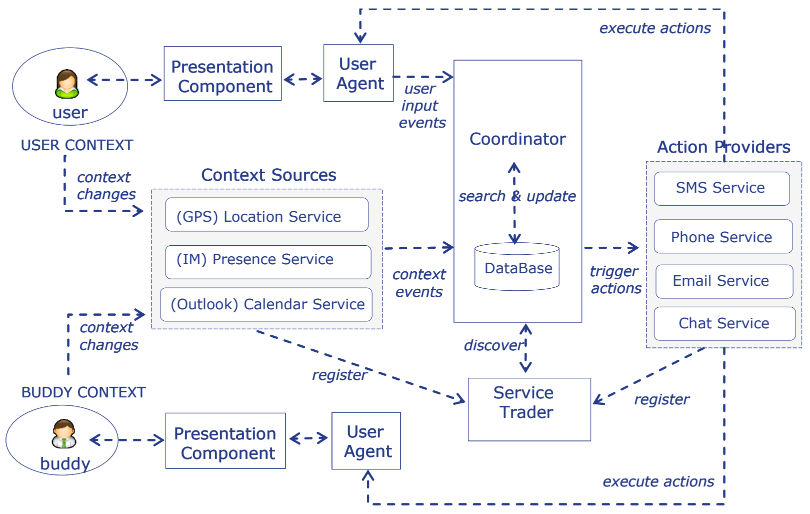


Figure 91 Reference architecture for context-aware mobile applications

Below we discuss the refinement of the Remove Buddy and Proximity functions according to this architecture.

8.2.1 Remove Buddy refinement

Figure 92 zooms into the details of the Remove Buddy function, which involves the User Agent, Coordinator and Database components. The status information handled by this function is the same as depicted in *Figure 89*, but assigned to the proper corresponding component of the reference architecture.

The Remove Buddy function in *Figure 92* consists of five *basic interaction patterns* (see Section 3.3), namely recurrent interactions between components of our reference architecture (two components per basic interaction pattern). These basic patterns are Remove Request, Remove Search, Remove Update, Remove Acceptance and Remove Rejection, and we composed them using some *sequence flow* elements and an *exclusive decision gateway* element in order to form the Remove Buddy function. Therefore, we consider the Remove Buddy function as a *composite interaction pattern*.

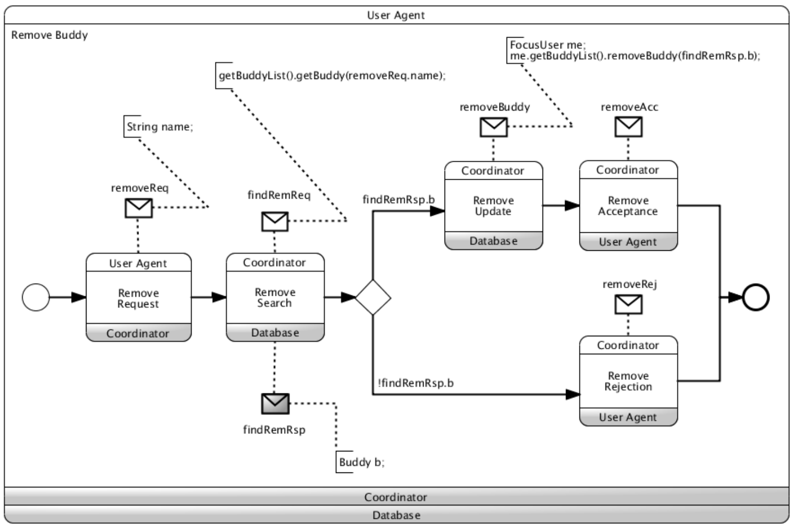


Figure 92 SDRM, Remove Buddy

Figure 92 represents the following behaviour:

1. In order to remove one of the user's buddies from the contact list, the user agent should inform the coordinator about the removal request. This is represented as a Remove Request pattern, in which the User Agent sends a removeReq message that contains the name of the buddy to be removed (String name) to the Coordinator.
2. In order to assure that the buddy to be removed is actually in the contact list of the user, the coordinator should retrieve the corresponding buddy's values from the database. This is represented as a two-way Remove Search basic interaction pattern, in which (a) the Coordinator sends to the Database an operation request findRemReq, (b) the Database performs this operation, and (c) the Database sends back the operation return value (Buddy b) to the Coordinator with a findRemReq message.
3. In case the search return value is not empty (findRemRsp.b condition), i.e., the buddy is actually in the contact list of the user, the coordinator should remove this buddy as requested by the user. This is represented as a one-way Remove Update basic interaction pattern, in which the Coordinator requests the removal of the buddy from the Database with a removeBuddy message. 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 requested buddy (removeAcc message).
4. In case the return value is empty (!findRemRsp.b condition), i.e., the buddy is not in the user's contact list, the coordinator should inform

the user about the impossibility of removing this buddy. This is represented as a one-way Rejection Response basic interaction pattern, in which the Coordinator informs the User Agent with a removeRej message that the removal request could not be fulfilled.

8.2.2 Proximity refinement

Figure 93 zooms into the details of the Proximity function, which involves the Coordinator, Context Source, Database and User Agent components. The context source is the component dedicated to sense changes in the user's context and provides the coordinator with context events. As discussed in Section 6.3.2, although there are several context sources distributed in the environment, we assume in *Figure 93* that only one context source communicates with the coordinator at a time, namely the context source that has sensed the event of interest. In order to avoid clogging the figure, we have not included the status information handled by these components. This information is the same as depicted in *Figure 90*, but assigned to the proper corresponding component of the reference architecture.

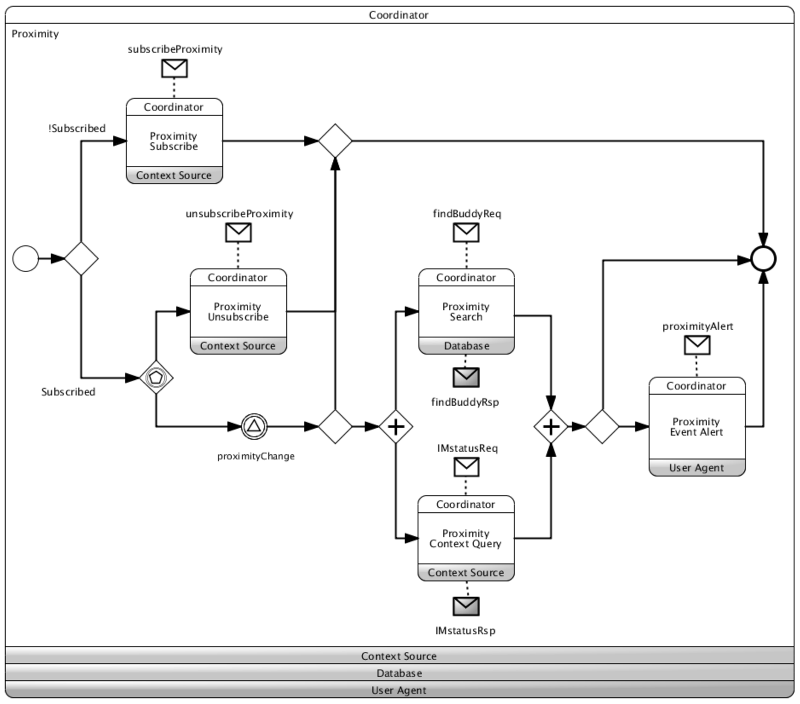


Figure 93 SDRM, Proximity

The Proximity function in *Figure 93* represents a composite interaction pattern that consists of six *basic interaction patterns* named Proximity Subscribe, Proximity Unsubscribe, Proximity Signal Event, Proximity Search, Proximity Context Query and Proximity Event Alert. *Figure 93* represents the following behaviour:

1. In order to receive proximity events the coordinator should subscribe to the context source for those events. This is represented with a decision controlled by a variable named Subscribed. In case the Subscribed variable is false, a Proximity Subscribe basic interaction pattern should be perfomed in order to subscribe the Coordinator to the Context Source. The subscribeProximity message sent by the Coordinator to the Context Source contains information that refers to the context model discussed in Section 6.1.2 (see *Figure 61*). Once the subscription is done, the Subscribed variable must be set to true.
2. In order to stop receiving proximity events the coordinator should unsubscribe to the context source for those events. This is represented with the Proximity Unsubscribe basic interaction pattern, in which the Coordinator stops the subscription to the Context Source. As a consequence of this unsubscription, the Subscribed variable must be set to false.
3. Whenever a proximity event takes place, the coordinator should be warned by the context source. This is represented with a proximityChange signal event generated eventually by the Context Source to the Coordinator, after which the Subscribed variable must be set to true. The actual implementation of the context subscription and notification mechanisms falls outside the scope of our platform-independent design.
4. In order to warn the user about the occurrence of a proximity event for a certain buddy, the coordinator should retrieve the name of this buddy from the database and present it to the user. This is represented with the Proximity Search interaction pattern between the Coordinator and Database, in which a findBuddyReq and findBuddyRsp message are exchanged that contain the name of this buddy.
5. In order to warn the user about the occurrence of a proximity event the coordinator should also synchronously retrieve from the context source the IM status of this buddy and assures that this value is "online". This is represented with the Proximity Context Query interaction pattern between the Coordinator and Context Source, in which an IMstatusReq and IMstatusRsp message are exchanged that contain this IM status value.
6. In case the IM status value is "online" a proximity alert should be generated for the user. This is represented with the one-way Proximity Event Alert basic interaction pattern, in which the Coordinator generates a proximityAlert notification message to the User Agent.
7. In case the IM status value is not "online", no proximity alert should be generated by the coordinator to the user agent. This is represented with the end of the Proximity function without warning the user.

8.3 SS to SDRM Refinement Transformation

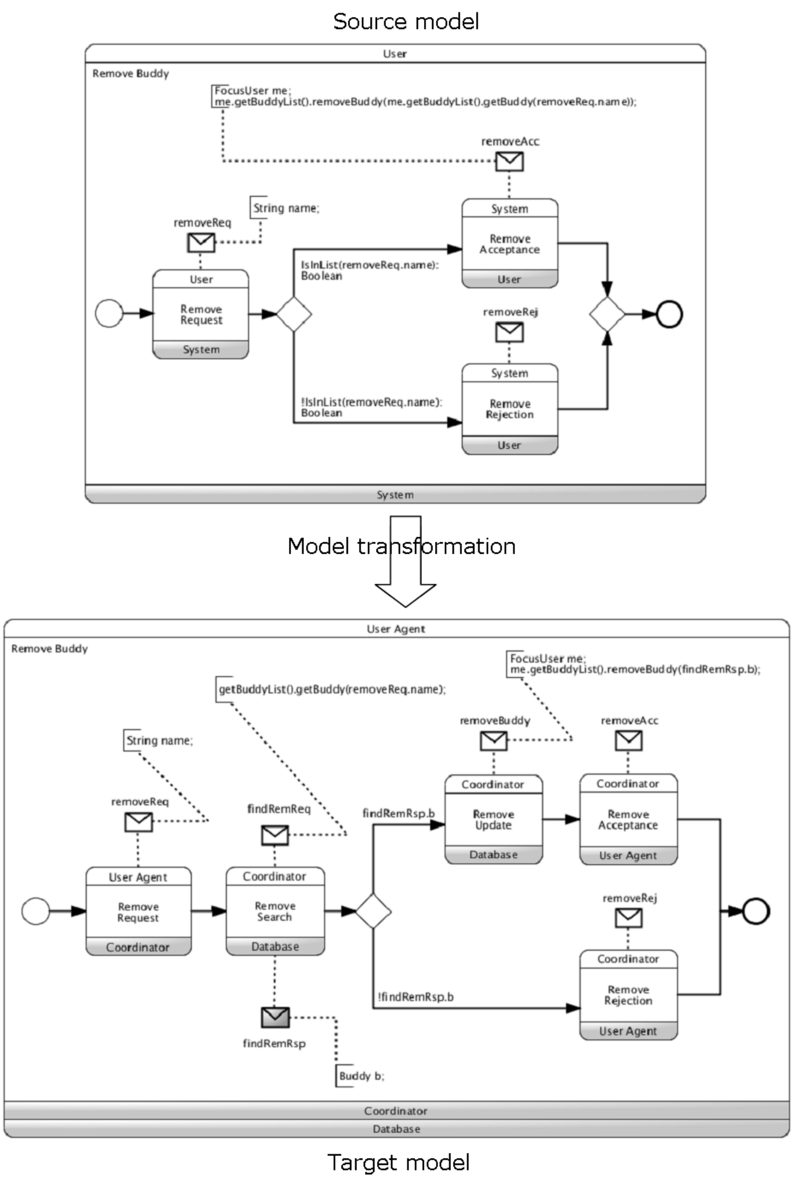
The $T_{SS,SDRM}$ refinement transformation from service functions in terms of interaction markers to more detailed functions in terms of interaction patterns takes as input the service specification (*SS*) discussed in Section 8.1, and generates the service design refined model (*SDRM*) discussed in Section 8.2.

8.3.1 Remove Buddy refinement transformation

Figure 94 shows the source model (see *Figure 89*) and target model (see *Figure 92*) for the $T_{SS,SDRM}$ Remove Buddy refinement transformation.

Figure 94 shows at a glance that the $T_{SS,SDRM}$ refinement transformation adds detail to the SDRM target model and, at the same time, preserves the behaviour structure of the SS source model. We have defined transformation rules in order to map *choreography task* elements in the SS source model, namely our interaction markers, onto refined *choreography task* elements in the SDRM target model, namely our interaction patterns. We have also defined transformation rules in order to map the SS behaviour structure in *Figure 94*, such as, for example, the *sequence flow* and *exclusive decision gateway* elements, onto corresponding behaviour structure elements in the SDRM target model. For the sake of readability, we do not present the complete set of transformation rules. *Figure 95* shows schematically the mappings that we have used to define a sub-set of these transformation rules, which we consider as the most significant for the purpose of this thesis.

Figure 94 Source (SS) and target (SDRM) models for the Remove Buddy refinement transformation



```
FocusUser me;
me.getBuddyList().removeBuddy(me.getBuddyList().getBuddy(removeReq.name));
```

```
getBuddyList().getBuddy(removeReq.name);
```

```
FocusUser me;
me.getBuddyList().removeBuddy(findRemRsp.b);
```

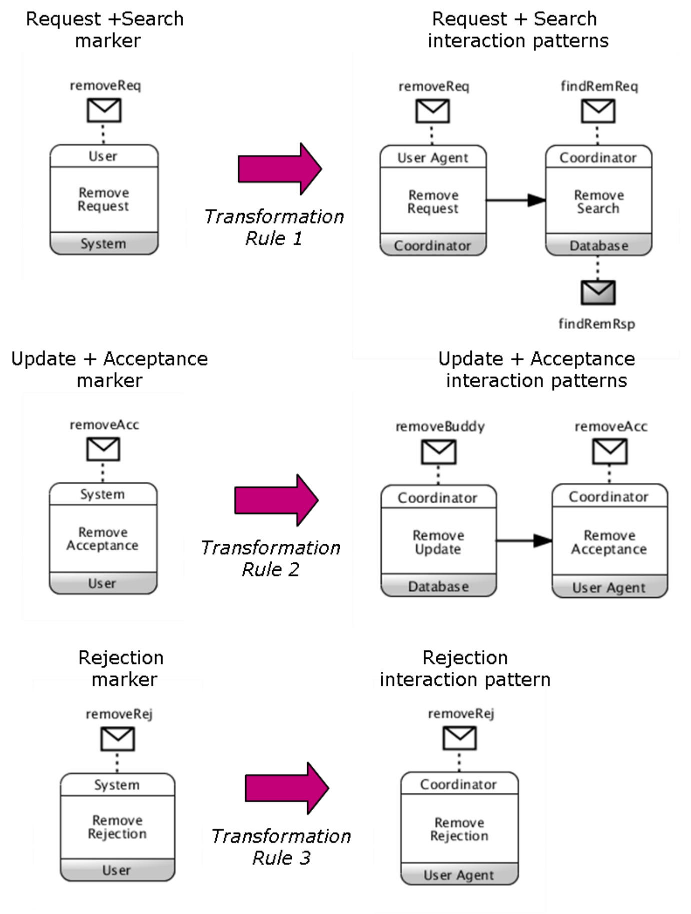


Figure 95 Remove Buddy: mappings for SStoSDRM transformation rules definition

The mappings in *Figure 95* relate SS interaction markers to SDRM interaction patterns according to the following transformation rules:

1. Transformation rule 1. Whenever a user makes a request, the user agent, which acts on behalf of the user, should forward this request to the coordinator. The coordinator is then responsible to perfom some task(s) in order to fulfil the user request. In case of the removal request, the first task of the coordinator consists of checking the information stored in the database in order to assure that the buddy to be removed is actually in the contact list of the user. In order to achieve this, *transformation rule 1* defines a mapping of an SS marker with name Remove Request onto the combination of a *request* interaction pattern with name Remove Request, and a *search* interaction pattern with name Remove Search. The SS Remove Request marker has a removeReq message

attached to the User participant, which is mapped onto the following messages in the SDRM target model:
- a removeReq message attached to the User Agent participant in the Remove Request interaction pattern
- a findRemReq message attached to the Coordinator participant in the Remove Search interaction pattern, and
- a findRemRsp message attached to the Database participant in the Remove Search interaction pattern.

For each *choreography task* element that is found in the SS source model with:
- name Remove Request
- participants User and System, and
- message removeReq attached to the User participant,

transformation rule 1 generates two *choreography task* elements with:
- names Remove Request and Remove Search, respectively,
- participants User Agent/Coordinator and Coordinator/Database, respectively, and
- messages removeReq, findRemReq, findRemRsp attached to the User Agent, Coordinator and Database participants, respectively.

Transformation rule 1 also generates a *sequence flow* element to connect the generated Remove Request and Remove Search *choreography task* elements in the SDRM target model.

2. Transformation rule 2. In case the buddy requested for removal is in the contact list of the user, the coordinator should update the user's contact list in the database by removing this buddy, and inform the user about the succesfull removal. In order to achieve this, *transformation rule 2* defines a mapping of an SS marker with name Remove Acceptance onto the combination of an *update* interaction pattern with name Remove Update, and an *acceptance* interaction pattern with name Remove Acceptance. The SS Remove Acceptance marker has a removeAcc message attached to the System participant, which is mapped onto the following messages in the SDRM target model:
   - a removeBuddy message attached to the Coordinator participant in the Remove Update interaction pattern, and
   - a removeAcc message attached to the Coordinator participant in the Remove Acceptance interaction pattern.

   For each *choreography task* element that is found in the SS source model with:
   - name Remove Acceptance
   - participants System and User, and
   - message removeAcc message attached to the System participant,

   transformation rule 2 generates two *choreography task* elements with:

- names Remove Update and Remove Acceptance, respectively
- participants Coordinator/Database and Coordinator/User Agent, respectively, and
- messages removeBuddy and removeAcc attached to the Coordinator participant.

Transformation rule 2 also generates a *sequence flow* element to connect the generated Remove Update and Remove Acceptance *choreography task* elements in the SDRM target model.

3. Transformation rule 3. In case the buddy requested for removal is not in the contact list of the user, the coordinator should inform the user about the impossibility of removing this buddy. In order to achieve this, *transformation rule 3* defines a mapping of an SS marker with name Remove Rejection onto a *rejection* interaction pattern with name Remove Rejection. The SS Remove Rejection marker has a removeRej message attached to the System participant, which is mapped onto a corresponding removeRej message attached to the Coordinator participant in the SDRM Remove Rejection interaction pattern. For each *choreography task* element that is found in the SS source model with name Remove Rejection, participants System and User, and message removeRej message attached to the System participant, transformation rule 3 generates a corresponding *choreography task* element with name Remove Rejection, participants Coordinator and User Agent, and message removeRej attached to the Coordinator participant.

8.3.2 Proximity refinement transformation

Figure 96 shows the source model (see *Figure 90*) and target model (see *Figure 93*) for the $T_{SS,SDRM}$ Proximity refinement transformation.

Analogously to the Remove Buddy transformation, for the Proximity refined transformation we also defined transformation rules in order to map *choreography task* elements in the SS source model, namely our interaction markers, onto refined *choreography task* elements in the SDRM target model, namely our interaction patterns. Moreover, we defined transformation rules in order to map the SS behaviour structure in *Figure 96*, such as, for example, *sequence flow* and *gateway* elements, onto corresponding behaviour structure elements in the SDRM target model. For the sake of readability, we do not present the complete set of transformation rules. *Figure 97* shows the mappings that we have used to define a sub-set of these transformation rules.

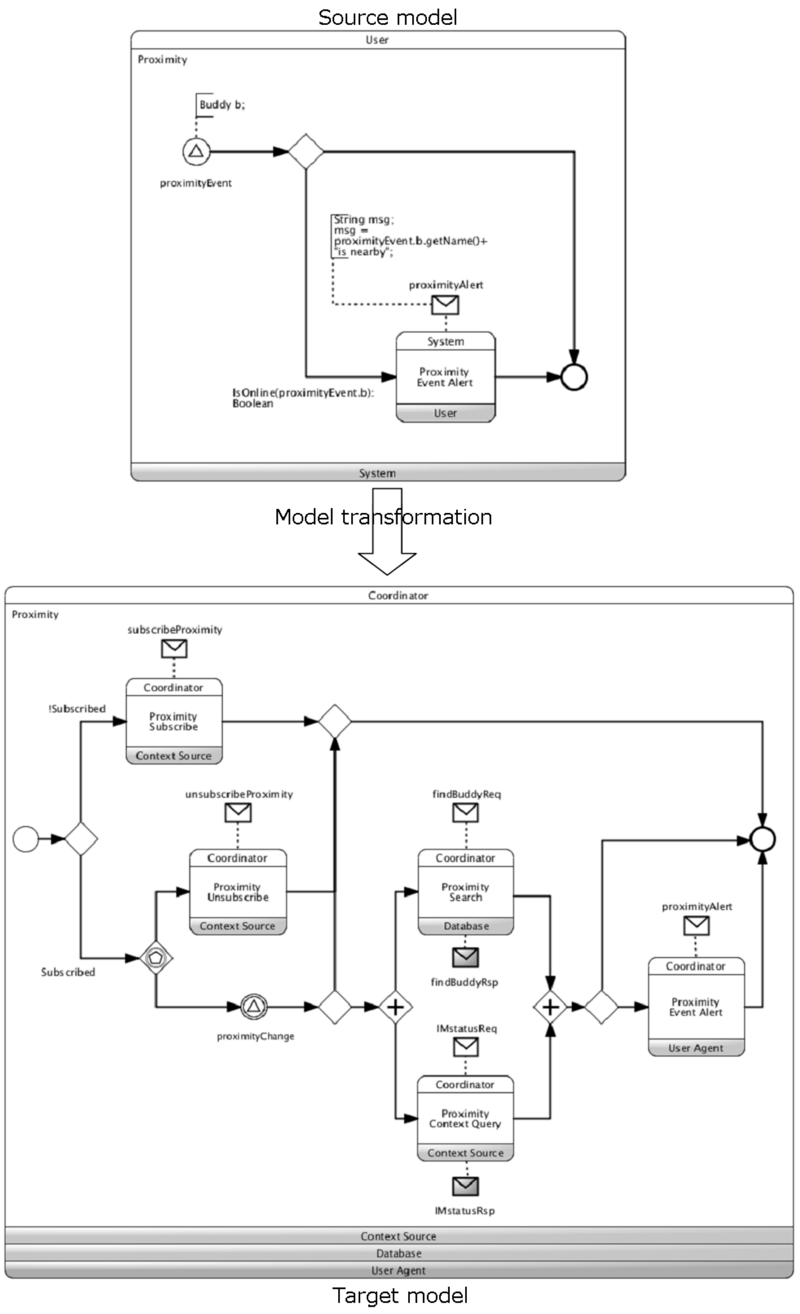


Figure 96 Source (SS) and target (SDRM) models for the Proximity refinement transformation

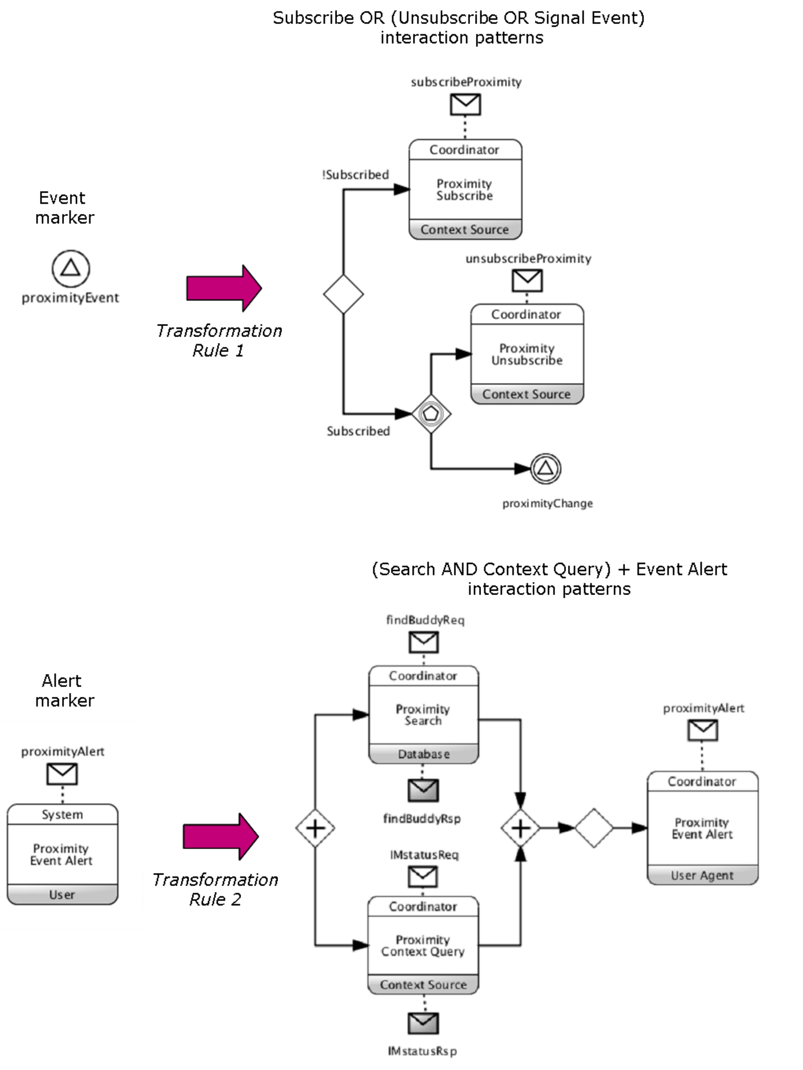


Figure 97 Proximity: mappings for SStoSDRM transformation rules definition

The mappings in *Figure 97* relate SS interaction markers to SDRM interaction patterns according to the following transformation rules:

1. Transformation rule 1. In order to receive proximity events the coordinator should subscribe to the context source for those events. As a consequence, whenever a proximity event takes place, the coordinator should be warned by the context source. Whenever the coordinator decides to stop receiving proximity events, it should unsubscribe to the context source for those events. In order to acheve this, *transformation rule 1* defines a mapping of an SS marker with name proximityEvent onto

the combination of a *subscribe* interaction pattern with name Proximity Subscribe, an *unsubscribe* interaction pattern with name Proximity Unsubscribe, and a *signal event* interaction pattern with name proximityChange. The SS proximityEvent marker is further mapped onto the following messages in the SDRM target model:

- a subscribeProximity message attached to the Coordinator participant in the Proximity Subscribe interaction pattern, and
- an unsubscribeProximity message attached to the Coordinator participant in the Proximity Unsubscribe interaction pattern.

For each *signal event* element that is found in the SS source model with name proximityEvent, *transformation rule 1* generates a corresponding *signal event* element with name proximityChange, and two *choreography task* elements matching the following pattern:

- names Proximity Subscribe and Proximity Unsubscribe, respectively
- participants Coordinator and Context Source, and
- messages subscribeProximity and subscribeProximity attached to the Coordinator participant.

Transformation rule 1 also generates an *exclusive decision gateway* element, an *event-based decision gateway* element, and the *sequence flow* elements that connect these *gateway* elements with the *choreography task* and *signal event* elements in *Figure 97*.

2. Transformation rule 2. In order to warn the user about the occurrence of a proximity event for a certain buddy, the coordinator should retrieve the name of this buddy from the database and present it to the user. Moreover, the coordinator should also synchronously retrieve from the context source the IM status of this buddy and assures that this value is "online". In case the IM status value is "online" a proximity alert should be generated for the user. In case the IM status value is not "online", no proximity alert should be generated by the coordinator to the user agent. In order to achieve this, *transformation rule 2* defines a mapping of an SS marker with name Proximity Event Alert onto the combination of a *search* interaction pattern with name Proximity Search, a *context query* interaction pattern with name Proximity Context Query, and an *event alert* interaction pattern with name Proximity Event Alert. The SS Proximity Event Alert marker has a proximityAlert message attached to the System participant, which is mapped onto the following five messages in the SDRM target model:
   - a findBuddyReq message attached to the Coordinator participant in the Proximity Search interaction pattern
   - a findBuddyRsp message attached to the Database participant in the Proximity Search interaction pattern

- an IMstatusReq message attached to the Coordinator participant in the Proximity Context Query interaction pattern
- a findBuddyRsp message attached to the Database participant in the Proximity Context Query interaction pattern, and
- a proximityAlert message attached to the Coordinator participant in the Proximity Event Alert interaction pattern.

For each *choreography task* element that is found in the SS source model matching the following pattern:

- name Proximity Event Alert
- participants System and User, and
- message proximityAlert message attached to the System participant,

transformation rule 2 generates three *choreography task* elements with:

- names Proximity Search, Proximity Context Query and Proximity Event Alert, respectively
- participants Coordinator/Database, Coordinator/Context Source, and Coordinator/User Agent, respectively, and
- the five messages mentioned above.

Transformation rule 2 also generates in the SDRM model a *parallel decision gateway* element, a *merge gateway* element, an *exclusive decision gateway* element, and the *sequence flow* elements that connect these gateways as depicted in *Figure 97*.

8.4 Service Design Component Model

The service design component model synthesises the interactions represented in the service design refined model (see Section 8.2) in the internal behaviour of the components involved in these interactions. In this way, each component at the SDCM level is characterised by an internal flow of activities that can be used for execution purposes while preserving the interactions with the other components prescribed by the SDRM level. Below we discuss the synthesis of the Remove Buddy and Proximity functions.

8.4.1 Remove Buddy synthesis

Figure 98 represents the Remove Buddy function as a BPMN collaboration diagram between the User Agent, the Coordinator and the Database components, which are represented as pools. Each of these pools contains a Remove Buddy sub-process that describes the set of internal activities performed within that specific component in order to fulfil the request of removing a buddy from the buddy list of the user. We represent our SDCM level from the perspective of the Coordinator component, since it orchestrates the other components of our reference architecture.

Therefore, our SDCM diagram consists of an orchestration from the perspective of the Coordinator component. The status information handled by these components is not shown in order to avoid clogging the figure. However, this information is the same as depicted in *Figure 92*.

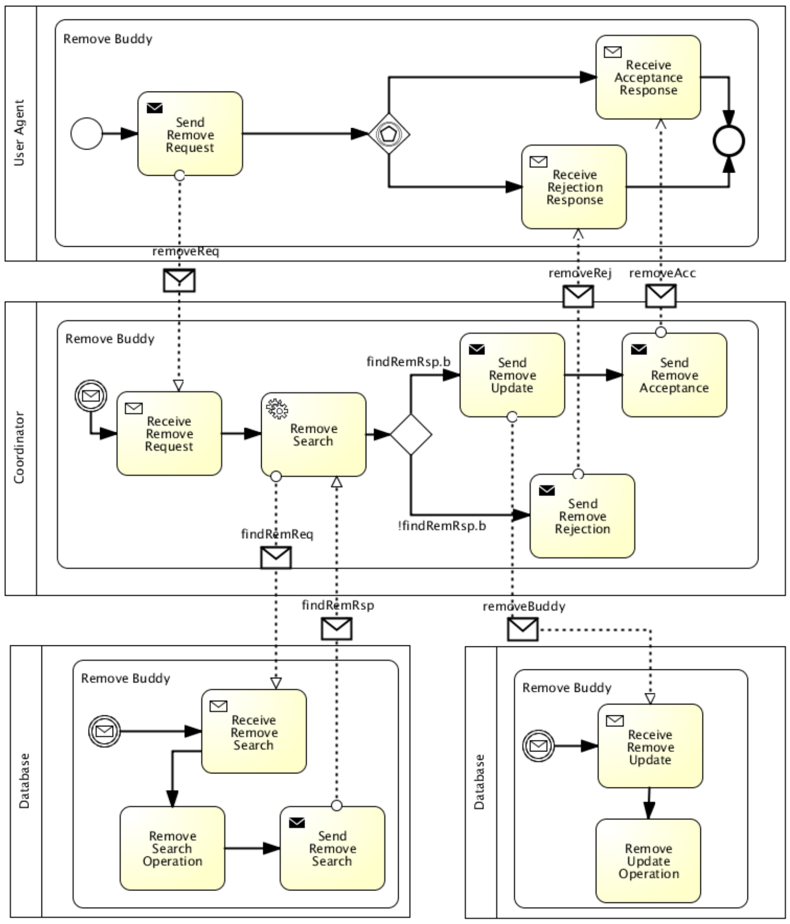


Figure 98 SDCM, Remove Buddy

In *Figure 98* we have identified five *basic executable patterns* (see Section 3.2), which are the Remove Request, Remove Search, Remove Update, Remove Acceptance and Remove Rejection. *Figure 98* represents the following behaviour:

1. In order to remove one of the user's buddies from the contact list, the user agent should inform the coordinator about the removal request. This is represented as a Remove Request pattern, in which the User Agent sends a removeReq message to the Coordinator with the Send Remove

Request task. This initiates the Remove Buddy sub-process of the Coordinator, which was waiting for a message in order to start.

2. Upon the reception of the removeReq message (Receive Remove Request task), the coordinator should retrieve the corresponding buddy's values from the database in order to assure that the buddy to be removed is in the contact list of the user. This is represented as a Remove Search pattern, which consists of an interaction between the Coordinator and the Database. In this interaction the Database receives a findRemReq message (Receive Remove Search task), performs an internal operation to fulfil this request (Remove Search Operation task), and sends a findRemRsp response message to the Coordinator (Send Remove Search task). Depending on the findRemRsp response, the Coordinator behaves as follows:
3. In case the findRemRsp response value is not empty (findRemRsp.b condition), i.e., the buddy is in the contact list of the user, the coordinator should remove this buddy as requested by the user. This is represented as a Remove Update pattern, in which the Coordinator sends a removeBuddy update message request to the Database (Send Remove Update task), which receives the request (Receive Remove Update task) and updates the data store with an internal operation (Remove Update Operation task). A removeAcc confirmation follows (Send Remove Acceptance task) from the Coordinator to the User Agent (Receive Remove Acceptance task), and the Remove Buddy function ends.
4. In case the the findRemRsp response value is empty (!findRemRsp.b condition), i.e., the buddy is not in the user's contact list, the coordinator should inform the user about the impossibility of removing this buddy. This is represented as a Remove Rejection pattern, in which the Coordinator sends a removeRej message response (Send Rejection Response task) to the User Agent (Receive Remove Rejection task).
5. In both cases, as indicated by the *event-based gateway* element in the User Agent pool in *Figure 98,* the User Agent waits for a removeAcc or removeRej message from the Coordinator, which is responsible of the acceptance or rejection decision.

8.4.2 Proximity synthesis

Figure 99 represents the Proximity function as a BPMN collaboration diagram between the Context Source, Coordinator, Database and User Agent components represented as pools. Each of these pools contains a Proximity sub-process that describes the set of internal activities performed within that specific component for a proximity event.

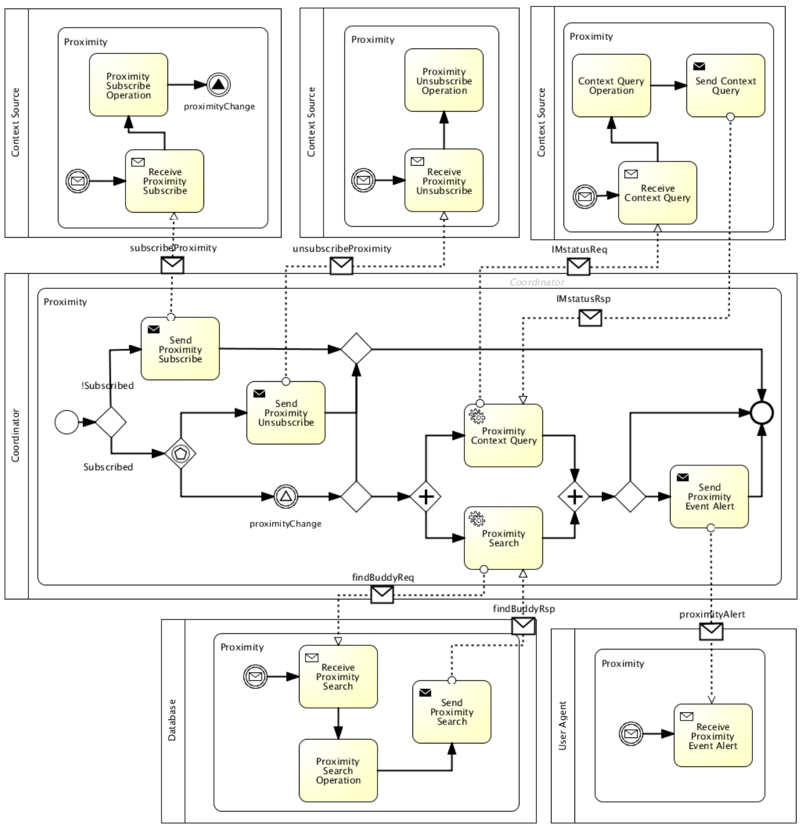


Figure 99 SDCM, Proximity

In *Figure 99* we have identified six basic executable patterns, which are Proximity Subscribe, Proximity Unsubscribe, Proximity Signal Event, Proximity Search, Proximity Context Query and Proximity Event Alert. *Figure 99* shows the following behaviour:

1. The Coordinator process initiates the Proximity function with a start event, and evaluates whether a subscription for the proximity event has already occurred or not depending on the value of the Subscribed control variable (see Section 8.2.2).
2. In case subscription is necessary (!Subscribed condition), the Send Proximity Subscribe task is performed by the Coordinator, which sends a subscriptionRequest message to an appropriate Context Source (Receive Proximity Subscribe task). An internal task named Proximity Subscribe Operation is performed by this Context Source in order to realise the subscription. As a consequence, a proximityChange signal event is thrown by the Context Source every time a proximity situation between the user and one of his buddies starts to hold.

3. In case subscription has already been performed (Subscribed condition), the Send Proximity Unsubscribe task can be performed by the Coordinator, or, alternatively, a proximityChange signal event thrown by the Context Source can be caught by the Coordinator. In the latter case, the Coordinator performs the Proximity Search and Proximity Context Query tasks in parallel.
4. In the Proximity Search task, the Database receives a findBuddyReq message (Receive Proximity Search task), performs an internal operation to fulfil this request (Proximity Search Operation task), and sends a findBuddyRsp message to the Coordinator (Send Proximity Search task).
5. In the Proximity Context Query task, the Context Source receives an IMstatusReq message (Receive Proximity Context Query task), performs an internal operation to fulfill the query (Proximity Context Query Operation task), and returns an IMstatusRsp message to the Coordinator (Send Proximity Context Query task) .
6. Afterwards, the Send Proximity Event Alert can be sent from the Coordinator (Send Proximity Event Alert task) to the User Agent (Receive Proximity Event Alert task) only in case the retrieved IM status of the buddy nearby the user has value "online".

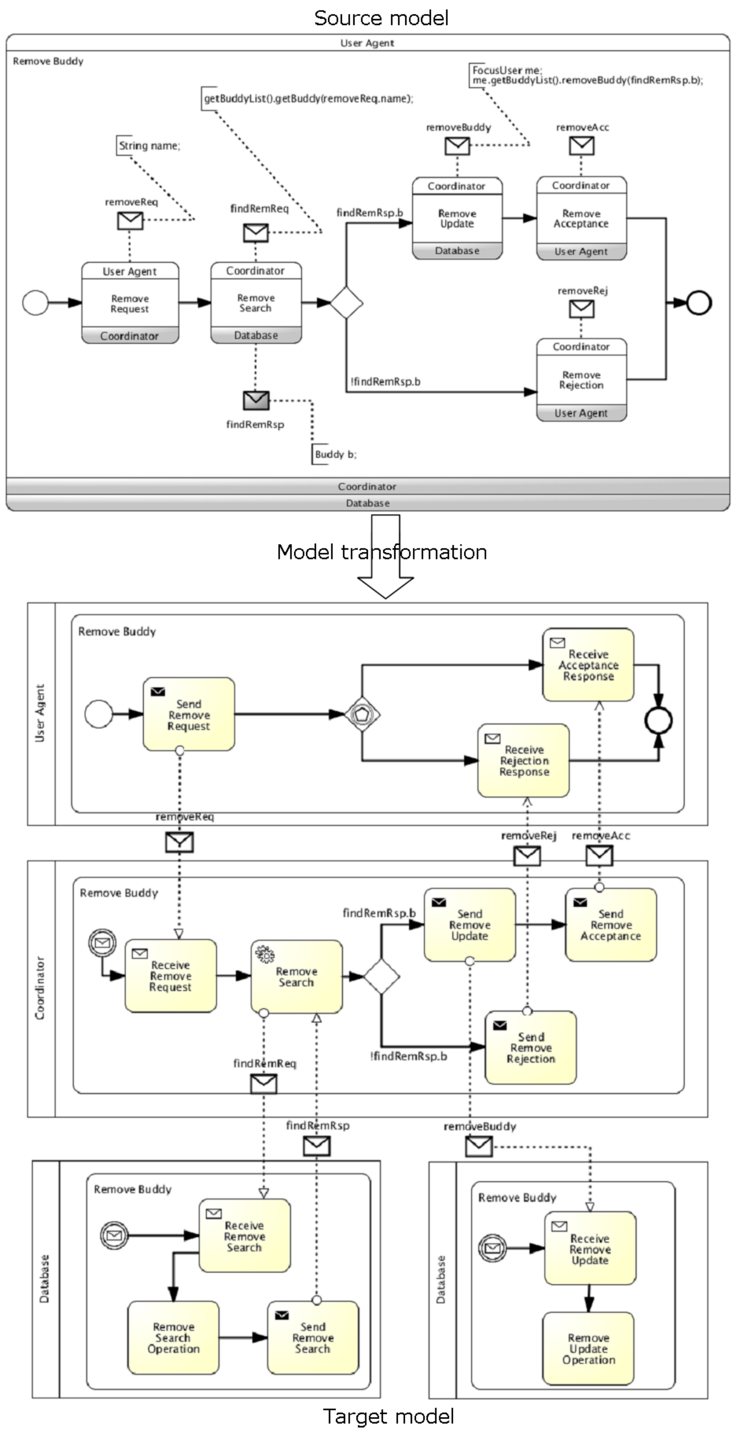
8.5 SDRM to SDCM Synthesis Transformation

The $T_{SDRM,SDCM}$ synthesis transformation from service functions in terms of interaction patterns to more detailed functions in terms of executable patterns takes as input the SDRM model discussed in Section 8.2, and generates the SDCM model discussed in Section 8.4.

8.5.1 Remove Buddy synthesis transformation

Figure 100 shows the source model (see *Figure 92*) and target model (see *Figure 98*) for the $T_{SDRM,SDCM}$ Remove Buddy synthesis transformation. *Figure 100* shows at a glance that the $T_{SDRM,SDCM}$ transformation adds detail to the SDCM target model in terms of the internal behaviour of the involved participants, preserving the behaviour structure of the SDRM source model. We have defined transformation rules in order to map *choreography task* elements in the SDRM source model, i.e., interaction patterns, onto *process task* elements in the SDRM target model, i.e., executable patterns. We have also defined transformation rules in order to map the SDRM behaviour structure in *Figure 100* onto corresponding behaviour structure elements in the SDCM target model. For the sake of readability, we do not present the complete set of transformation rules. *Figure 101* shows the mappings that we have used to define a sub-set of these transformation rules which we consider as the most significant for the purpose of this thesis.

Figure 100 Source (SDRM) and target (SDCM) models for the Remove Buddy synthesis transformation



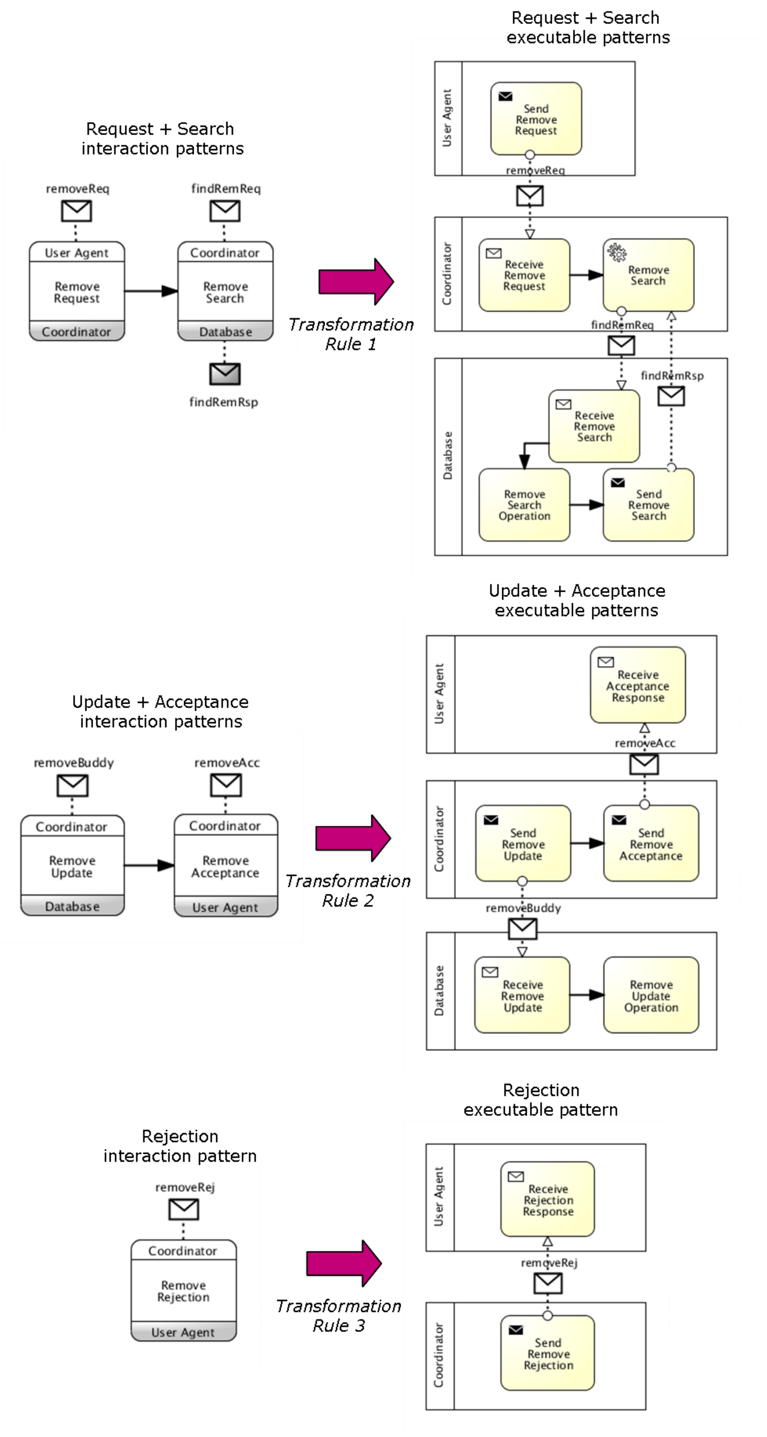


Figure 101 Remove Buddy: mappings for SDRMtoSDCM transformation rules definition

The mappings in *Figure 101* relate SDRM interaction patterns to SDCM executable patterns as follows:

1. <u>Transformation rule 1</u>. Whenever a user makes a request, the user agent, which acts on behalf of the user, should forward this request to the coordinator. The coordinator is then responsible to perfom some task(s) in order to fulfil the user request. In case of the removal request, the first task of the coordinator consists of checking the information stored in the database in order to assure that the buddy to be removed is actually in the contact list of the user. In order to achieve this, *transformation rule 1* defines a mapping of the following SDRM interaction patterns:

– an SDRM *request* with name Remove Request and
– an SDRM *search* with name Remove Search,

onto the following SDCM executable patterns:

– the sequence of a Send Remove Request *process task* element in the User Agent pool and a Receive Remove Request *process task* in the Coordinator pool, and
– the sequence of a Remove Search *process task* element in the Coordinator pool, a Receive Remove Search *process task* element in the Database pool, a Remove Search Operation *process task* element in the Database pool, and a Send Remove Search *process task* element in the Database pool.

The following messages in the SDRM source model are mapped onto equivalent messages in the SDCM target model:

– the removeReq message attached to the User Agent participant in the Remove Request interaction pattern,
– the findRemReq message attached to the Coordinator participant in the Remove Search interaction pattern, and
– the findRemRsp message attached to the Database participant in the Remove Search interaction pattern.

Transformation rule 1 generates the SDCM elements mentioned above whenever two *choreography task* elements are found in the SDRM source model matching the following pattern:

– names Remove Request and Remove Search, respectively
– participants User Agent/Coordinator and Coordinator/Database, respectively, and
– messages removeReq, findRemReq, findRemRsp attached to the User Agent, Coordinator and Database participants, respectively.

Transformation rule 1 also generates the *sequence flow* elements as depicted in *Figure 101* in order to connect the generated *process task* elements in the SDCM target model.

2. Transformation rule 2. In case the buddy requested for removal is in the contact list of the user, the coordinator should update the user's contact list in the database by removing this buddy, and inform the user about the succesfull removal. In order to achieve this, *transformation rule 2* defines a mapping of the following SDRM interaction patterns:
   - an SDRM *update* with name Remove Update and
   - an SDRM *acceptance* with name Remove Acceptance,

   onto the following SDCM executable patterns:
   - the sequence of a Remove Update *process task* element in the Coordinator pool, a Receive Remove Update *process task* element in the Database pool, and a Remove Update Operation *process task* element in the Database pool, and
   - the sequence of a Send Remove Acceptance *process task* element in the Coordinator pool, and a Receive Remove Acceptance *process task* in the User Agent pool.

   The following messages in the SDRM source model are mapped onto equivalent messages in the SDCM target model:
   - the removeBuddy message attached to the Coordinator participant in the Remove Update interaction pattern, and
   - the removeAcc message attached to the Coordinator participant in the Remove Acceptance interaction pattern.

   Transformation rule 2 generates the SDCM elements mentioned above whenever two *choreography task* elements are found in the SDRM source model matching the following pattern:
   - names Remove Update and Remove Acceptance, respectively
   - participants Coordinator/Database and Coordinator/User Agent, respectively, and
   - messages removeBuddy and removeAcc attached to the Coordinator participant, respectively.

   Transformation rule 2 also generates the *sequence flow* elements as depicted in *Figure 101* in order to connect the generated *process task* elements in the SDCM target model.

3. Transformation rule 3. In case the buddy requested for removal is not in the contact list of the user, the coordinator should inform the user about the impossibility of removing this buddy. In order to achieve this, *transformation rule 3* defines a mapping of an SDRM *rejection* interaction pattern with name Remove Rejection onto a corresponding SDCM executable pattern with name Remove Rejection. The SDRM Remove Rejection interaction pattern has a removeRej message attached to the Coordinator participant, which is mapped onto a corresponding removeRej message between the Coordinator and User Agent pools in the SDCM Remove Rejection executable pattern. For each *choreography task* element

that is found in the SDRM source model matching the following pattern:

- name Remove Rejection
- participants Coordinator and User Agent, and
- message removeRej message attached to the Coordinator participant,

transformation rule 3 generates two corresponding *process task* elements with:

- names Send Remove Rejection and Receive Remove Rejection, respectively
- pools Coordinator and User Agent, and
- message removeRej between the Coordinator and User Agent pools.

8.5.2 Proximity synthesis transformation

Figure 102 shows the source model (see *Figure 93*) and target model (see *Figure 99*) for the $T_{SDRM,SDCM}$ Proximity synthesis transformation. *Figure 102* shows at a glance that the Proximity $T_{SDRM,SDCM}$ transformation adds detail to the SDCM target model in terms of the internal behaviour of the involved participants and, at the same time, preserves the behaviour structure of the SDRM source model. We have defined transformation rules in order to map *choreography task* elements in the SDRM source model, namely our interaction patterns, onto *process task* elements in the SDRM target model, namely our executable patterns. We have also defined transformation rules in order to map the SDRM behaviour structure in *Figure 102*, such as, for example, *sequence flow* and *exclusive decision gateway* elements, onto corresponding behaviour structure elements in the SDCM target model. For the sake of readability, we do not present the complete set of transformation rules. *Figure 103* shows the mappings that we have used to define a sub-set of these transformation rules, which we consider the most significant for the purpose of this thesis.

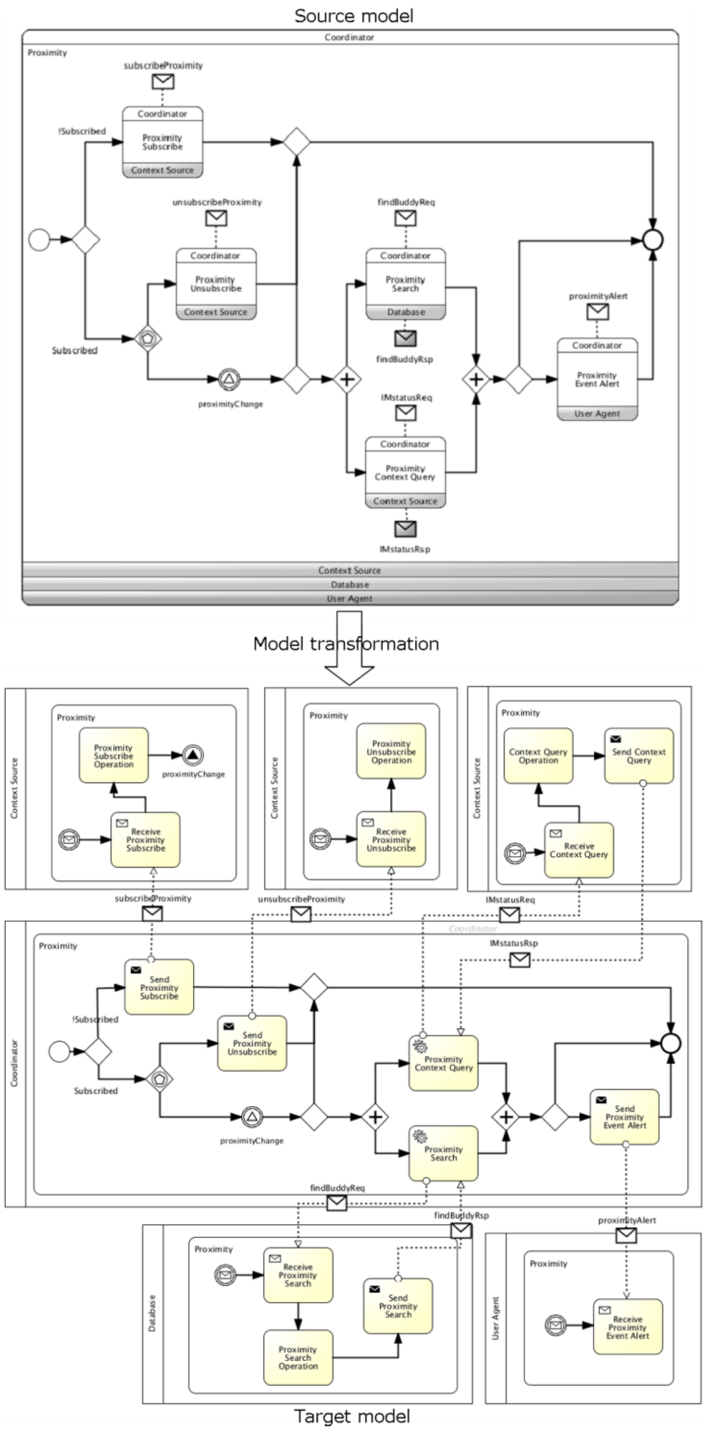


Figure 102 Source (SDRM) and target (SDCM) models for the Proximity synthesis transformation

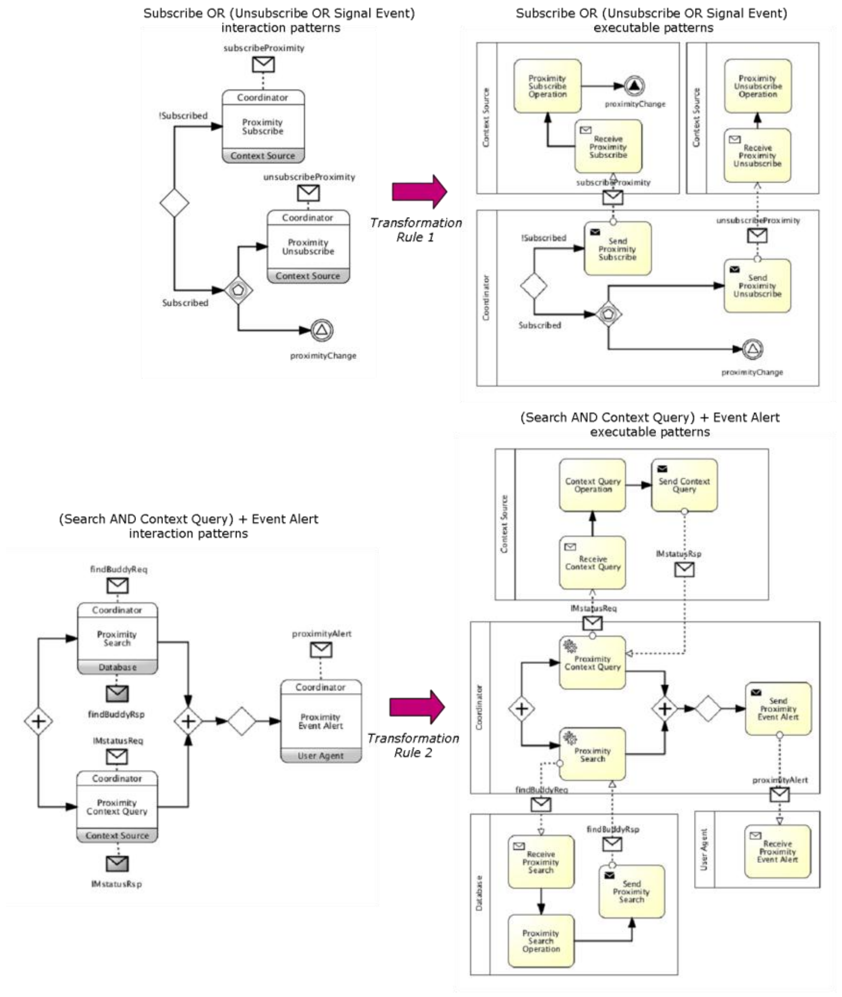

Subscribe OR (Unsubscribe OR Signal Event) interaction patterns
subscribeProximity
Coordinator
Proximity Subscribe
Context Source
!Subscribed
unsubscribeProximity
Proximity Unsubscribe
Subscribed
proximityChange
Transformation Rule 1
Subscribe OR (Unsubscribe OR Signal Event) executable patterns
Context Source
Proximity Subscribe Operation
Receive Proximity Subscribe
Proximity Unsubscribe Operation
Receive Proximity Unsubscribe
Send Proximity Subscribe
Send Proximity Unsubscribe
(Search AND Context Query) + Event Alert interaction patterns
findBuddyReq
Proximity Search
Database
findBuddyRsp
lMstatusReq
Proximity Context Query
lMstatusRsp
proximityAlert
Proximity Event Alert
User Agent
Transformation Rule 2
(Search AND Context Query) + Event Alert executable patterns
Context Query Operation
Send Context Query
Receive Context Query
Receive Proximity Search
Proximity Search Operation
Send Proximity Search
Send Proximity Event Alert
Receive Proximity Event Alert

Figure 103 Proximity: mappings for SDRMtoSDCM transformation rules definition

Since we assume that the mappings shown in *Figure 103* are similar to the mappings we have used to define the transformation rules presented so far in this chapter, we refrain from explaining these mappings.

8.6 Discussion

In this chapter, we have used BPMN to represent the source and target models for the refinement and synthesis transformations at the PIM level of our methodology. We implemented these transformations using the ATL transformation language. We did not discuss ATL transformation rules and language-specific constructs, but we introduced and justified in a language-independent way the mappings that we have used to implement these transformations.

The BPMN notation used in this chapter provided us with the expressiveness necessary to create three levels of behaviour models, namely SS, SDRM and SDCM levels. Although BPMN required a steep learning curve, this was balanced by the benefit of using only one language throughout the entire PIM design process. In this way, we had to handle only the BPMN metamodel when implementing the refinement and synthesis transformations, since all the SS, SDRM and SDCM models conform to the same metamodel. The mechanisms to collapse modelling elements provided by BPMN, such as collapsed sub-choreographies and sub-processes, helped us in mastering model size when the examples became more complex. Moreover, BPMN was suitable to represent our basic patterns of behaviours, and also the combinations of these basic patterns in more complex behaviours (composite patterns). Finally, since BPMN is widely adopted both in academia and industry, the use of this notation can be beneficial to make our methodology available to more people.

The availability of an Ecore version of the BPMN metamodel allowed us to define transformation rules in ATL for our *SStoSDRM refinement* and *SDRMtoSDCM synthesis* transformations and exectute these rules in an Eclipse-based envioronment using the ATL engine. The Eclipse environment has been our favorite choice as development platform, since it is an integrated environment in which we could both edit our BPMN models and realise our transformations either with the ATL engine or, alternatively, the medini QVT engine (see Section 6.4). However, we encountered some practical problems concerning Eclipse-based tool support for BPMN. The current version BPMN 2.0 is not supported yet by all currently available BPMN tools. For example, at the moment of writing, the BPMN modeller for the Eclipse platform was available only for BPMN version 1.2. Therefore, we used the Signavio/Oryx editor [134] to create the BPMN models presented in this thesis and exported these models to the Eclipse environment in order to use them as source and target models for the ATL transformation engine. The Signavio/Oryx editor is a process modelling platform that supports BPMN 2.0 and is freely available for academic use.

Chapter 9

Case Study

This chapter presents a case study that applies the PIM behaviour refinement and synthesis transformations implemented in Chapters 6 to 8 to the realisation of a running prototype at the PSM level that uses BPEL, UDDI and web services as target technology.

This chapter is organised as follows: Section 9.1 gives an overview of the case study, which realise the *contact buddy* function of the Live Contacts application, Section 9.2 presents our PIM behaviour models, namely *service specification (SS)*, *service design refined model (SDRM)* and *service design component model (SDCM)*, Section 9.3 discusses a platform-specific framework in which components of our reference architecture are mapped onto BPEL, UDDI and web services target technologies, Section 9.4 presents a PSM prototype that implements the PIM design using this framework, Section 9.5 discusses the *PIMtoPSM* transformation focusing on the coordinator component of our reference architecture, and, finally, Section 9.6 discusses the lesson learned with this case study.

9.1 Overview

In order to demonstrate that our PIM behaviour refinements can be applied to generate implementations at the PSM level, i.e., the applicability of our approach by means of a running application, we considered the following scenario based on the *contact buddy* function of the Live Contacts application:

"A user wants to contact one of his buddies with a specific communication means, such as SMS, Phone, Chat or E-mail. Therefore, the user provides the application with the name of this buddy and the communication means to be used. In order to fulfil the user request, the coordinator has to retrieve the contact details of the buddy from the buddy list of the user in the database, and discover a proper service in the service trader

according to the desired communication means. Once the coordinator has retrieved contact details of the buddy and the endpoint location of the communication service, it can forward this information to the user agent, which is finally able to invoke the proper service and put the user in communication with the desired buddy".

The case study starts modelling this scenario at the PIM level. In principle, any of the three solutions for PIM behaviour modelling and transformations proposed in Chapters 6, 7 and 8, respectively, could be used to model and realise these PIM transformations. Due to space limitations, we have chosen to show only the BPMN solution employed in Chapter 8, since BPMN is the most popular language among the proposed ones. The case study continues with the selection of the technology that we have chosen to realise our PSM design and the implementation of a prototype. *Figure 104* shows an overview of the case study.

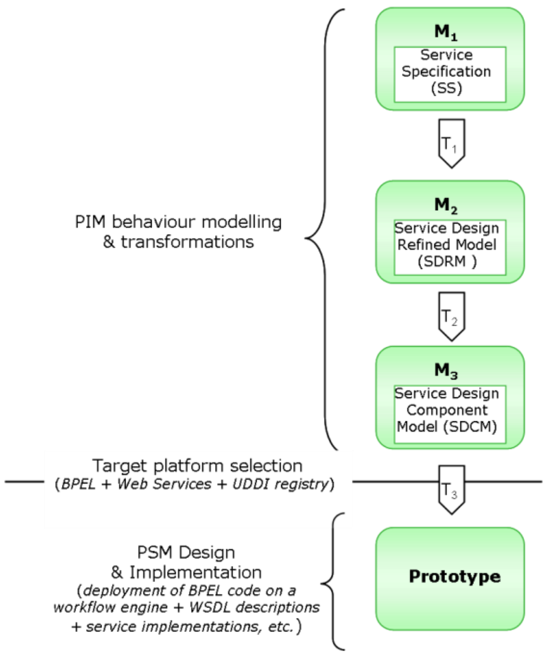

M1
Service Specification (SS)
T1
M2
Service Design Refined Model (SDRM)
T2
M3
Service Design Component Model (SDCM)
PIM behaviour modelling & transformations
Target platform selection
(BPEL + Web Services + UDDI registry)
T3
PSM Design & Implementation
(deployment of BPEL code on a workflow engine + WSDL descriptions + service implementations, etc.)
Prototype

Figure 104 Case study overview

9.2 PIM Design

This Section presents the models M_1, M_2 and M_3 shown in *Figure 104*. The model M_1 should be created by the application designer in collaboration with the user using interaction markers (see Section 3.5). The models M_2

and M_3 can be automatically generated executing the transformations T_1 and T_2 implemented in Chapter 8.

9.2.1 Service Specification

In the service specification phase of the design, we aim at composing the behaviour of the application under development by using *interaction markers* as placeholders for more concrete interactions at lower abstraction levels. *Figure 105* shows the service specification for the Contact Buddy function that is used in the Live Contacts application (see Section 6.1) when a user wants to contact one of his buddies with a specific communication means, such as SMS, Phone, Chat or E-mail. In order to establish this contact, the user should provide the system with the name of this buddy and the communication means to be used (contactReq message). As a consequence, the system reacts by opening the desired communication channel.

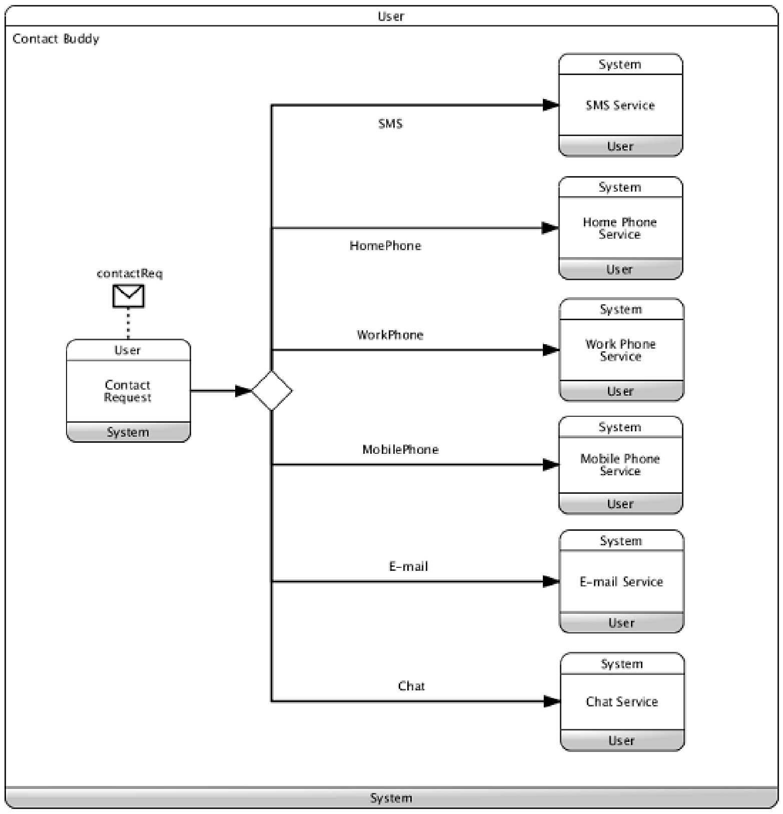


Figure 105 Contact buddy, SS

9.2.2 Service Design Refined Model

Figure 106 shows the service design refined model that refines the Contact Buddy function in *Figure 105* into a more concrete behaviour in terms of *basic interaction patterns*. The Contact Buddy function involves some components of our reference architecture, namely the user agent, the coordinator, the database, the service trader, and some action providers, which are the SMS, Phone, Chat and E-mail services.

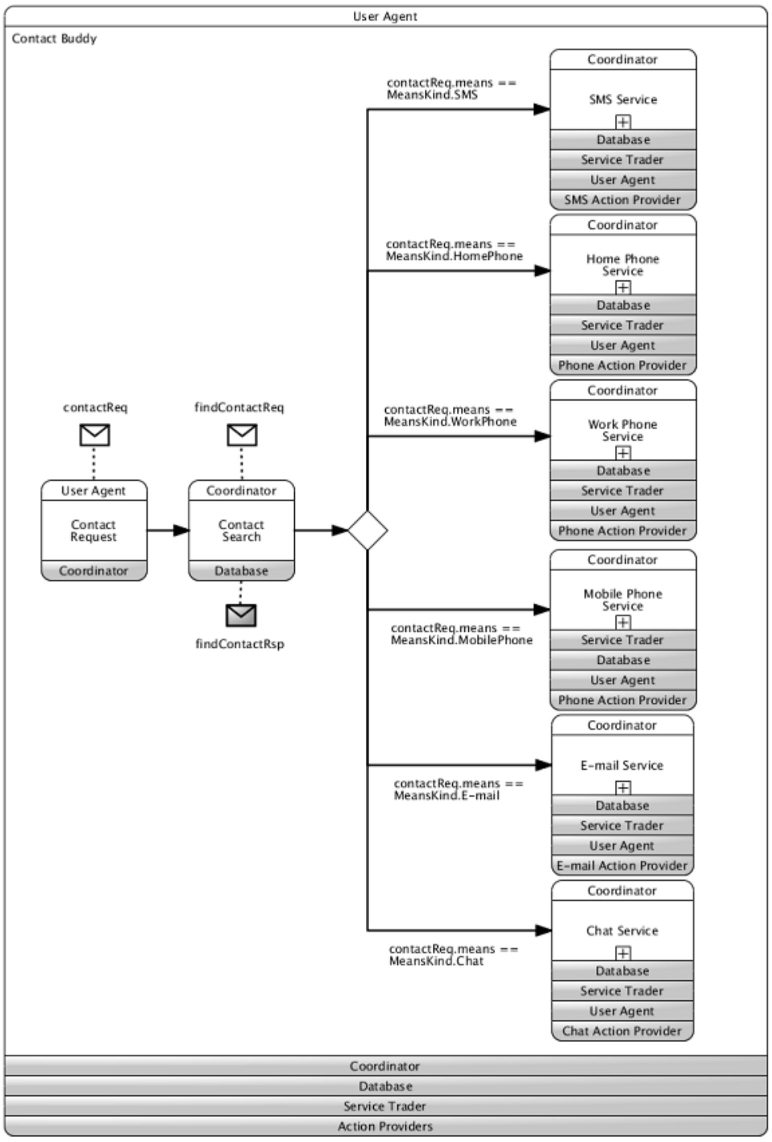


Figure 106 Contact buddy, SDRM

Figure 106 shows that the Contact Buddy function starts with a Contact Request from the User Agent to the Coordinator, which we classified as a (one-way) *request* basic interaction pattern. A Contact Search task follows, in which the Coordinator sends a findContactReq message to the Database and receives back a findContactRsp message. We classified this task as a (two-way) *search* basic interaction pattern. After the Contact Search pattern, the Coordinator selects the proper communication channel according to the user preferences. The details of the SMS Service are shown in *Figure 107*.

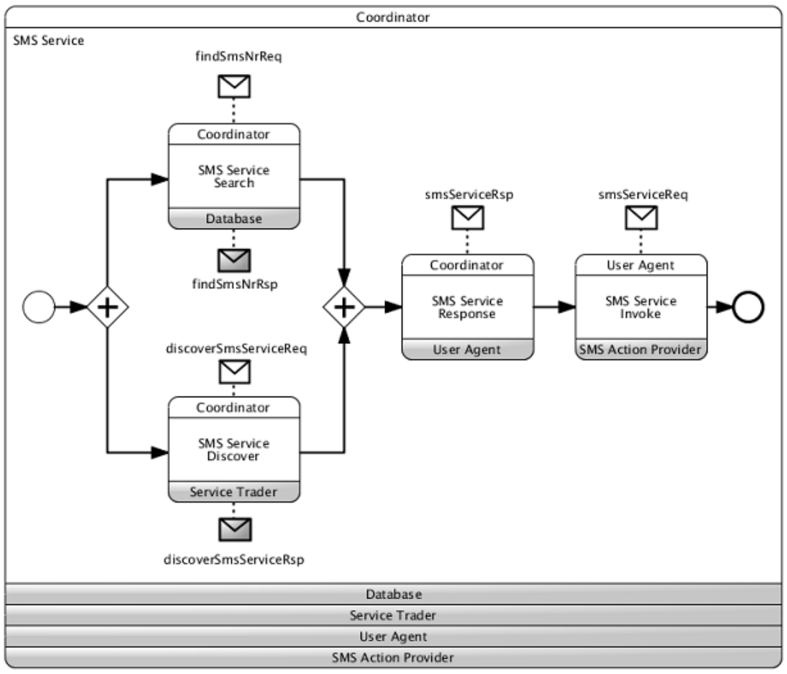


Figure 107 SMS service, SDRM

Figure 107 shows the following behaviour:

- in case the SMS Service is selected, the tasks SMS Service Search and SMS Service Discover are performed in parallel. These tasks correspond to our *search* and *discover* basic interaction patterns, respectively.
- In the SMS Service Search interaction pattern the Coordinator retrieves the SMS number of the contact from the Database with the findSmsNrReq and findSmsNrReq message exchange.
- In the SMS Service Discover pattern the Coordinator sends a discoverSmsServiceReq message to the Service Trader to indicate the service type to discover, namely "sms" in the given example, and the Service Trader returns the endpoint location of this service using a discoverSmsServiceRsp message.

– Once both the service discovery and the database retrieval are concluded, the Coordinator sends an SMS Service Response to the User Agent using an smsServiceRsp message that contains the information necessary to invoke the SMS service, i.e., the contact details of the buddy and the endpoint location of the service. This task corresponds to a *response* basic interaction pattern. In this way, the User Agent is able to invoke the Sms Action Provider and provide it with the necessary input, which is the mobile number to which the SMS message should be sent. This is represented by the Sms Service Invoke task, which we have classified as an *invoke* basic interaction pattern. We assume that any further information, such as the text of the SMS, should be provided directly by the user to the action provider, since this is part of the detailed behaviour of the action provider components, which is out of the scope of this thesis.

9.2.3 Service Design Component Model

Figure 108 shows the service design component model that synthesises the Contact Buddy function depicted in *Figure 106* and *Figure 107* into a more concrete function in terms of *basic executable patterns*. This model represents the executable behaviour of the coordinator component and its collaboration with the other components involved in the Contact Buddy function. These components are represented by pools in *Figure 108* and each of these pools contains a Contact Buddy sub-process instance. In order to simplify the model without loss of clarity, we only show in *Figure 108* the option in which the Coordinator has selected "sms" as preferred communication means.

Figure 108 Contact buddy, SDCM

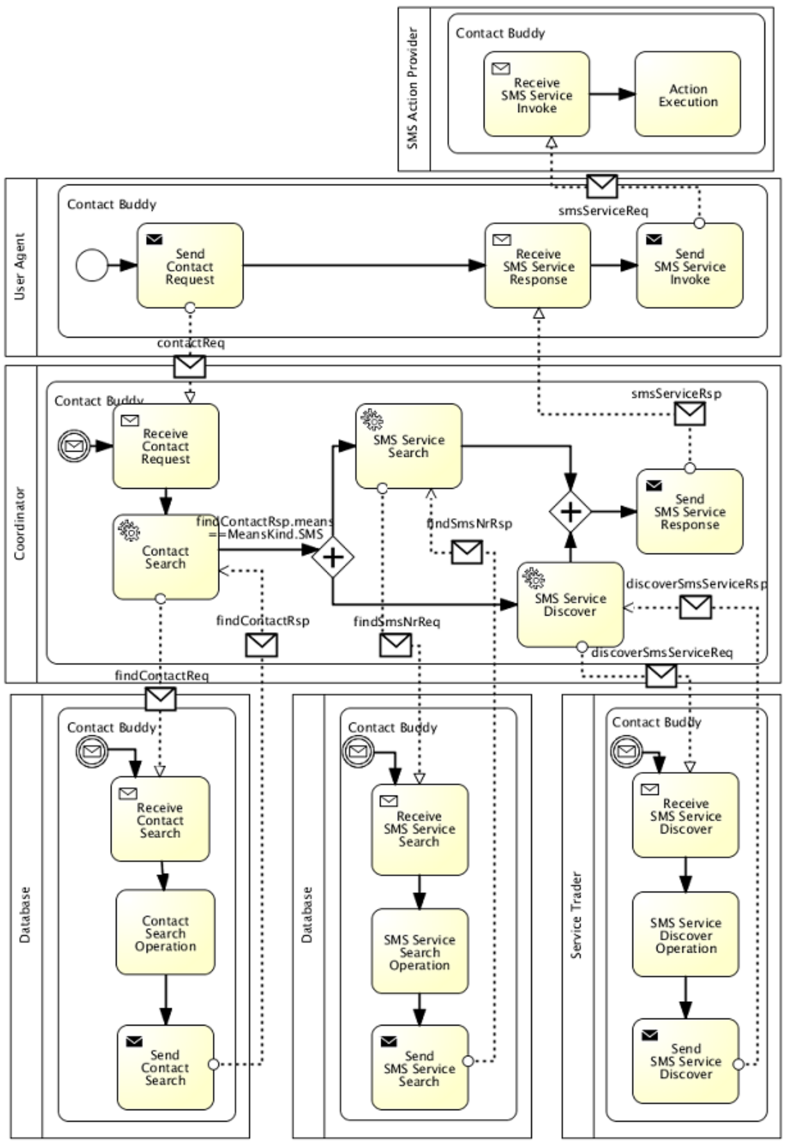


Figure 108 shows the following behaviour:

– the User Agent process initiates the Contact Buddy function with a start event. This is followed by a Send Contact Request task, in which the User Agent sends a contactReq message to the Coordinator. This initiates the Remove Buddy sub-process of the Coordinator, which is waiting for a message in order to start.

- Upon the reception of the contactReq message (Receive Contact Request task), the Coordinator performs a Contact Search task, which consists of retrieving the buddy to be called from the Database. In order to achieve this, the Database receives a findContactReq message from the Coordinator (Receive Contact Search task), performs an internal operation in order to get the expected buddy (Contact Search Operation task) and sends the response back to the Coordinator (Send Contact Search task) with a findContactRsp message.
- Afterwards, the Coordinator evaluates the value of the contact means previously provided by the User Agent in the contactReq message. Since we assume in the given example that this value is "sms", the Coordinator performs an SMS Service Discover task, which consists of retrieving the endpoint location of an SMS service from the Service Trader with the discoverSmsServiceReq and discoverSmsServiceRsp message exchange. In order to achieve this, the Service Trader receives the discoverSmsServiceReq message (Receive Sms ServiceDiscover task), performs an internal operation in order to discover the expected service (Sms Service Discover Operation task) and sends the discoverSmsServiceRsp message back to the Coordinator (Send Sms Service Discover task).
- Concurrently to this SMS Service Discover task, the Coordinator performs the SMS Service Search task, which consists of retrieving the phone number of the buddy to be called from the Database with the findSmsNrReq and findSmsNrRsp message exchange.
- Once both the endpoint location of an appropriate SMS service (discoverSmsServiceRsp message) and the phone number to contact the buddy (findSmsNrRsp message) are available, an smsServiceRsp message follows (Send Sms Service Response task) from the Coordinator to the User Agent (Receive Sms Service Response task).
- The User Agent uses the service endpoint information contained in the smsServiceRsp message to invoke the SMS Action Provider (Send Sms Service Invoke task). The SMS Action Provider receives the buddy phone number in the smsServiceReq message (Receive Sms Service Invoke task). Since we assume that any further information, such as the text of the SMS message, should be provided directly by the user to the SMS Action Provider, the Contact Buddy function in *Figure 108* ends with the Action Execution task performed by the SMS Action Provider.

We consider the SDCM model in *Figure 108* as the final artefact of our PIM level design. This model has in principle enough technical details for simulation, but technical information for deployment is still missing. Therefore, this SDCM model should be used as input for the PSM design and completed with the missing information before it can be deployed on some execution engine.

9.3 Technology Selection and PSM Design

In order to realise a PSM prototype that implements the scenario in Section 9.1, we have first designed a platform-specific framework in which components of our reference architecture are mapped onto target technologies. The same framework can be used with different scenarios. This framework is based on the Service-Oriented Architecture principles described in Section 2.2. We realised the coordinator as a BPEL process exposed as a web service to all the other components of the architecture. These components provide and possibly use services that are orchestrated by the coordinator BPEL process. *Figure 109* shows the framework. considered in the PSM design.

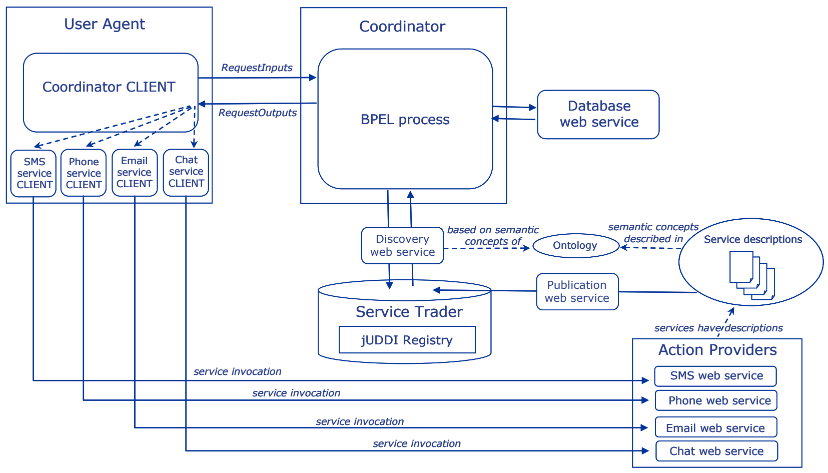
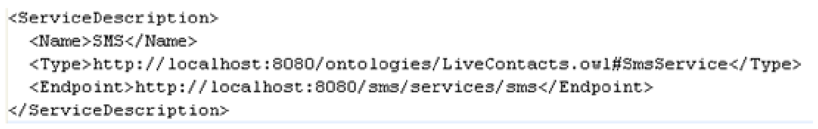


Figure 109 Platform-specific design framework

Figure 109 shows that the Coordinator BPEL process receives some inputs (RequestInputs) from a Coordinator client that should be implemented in the User Agent. These inputs instantiate a new Coordinator BPEL process. In our *contact buddy* prototype, the inputs are the name of the buddy and the preferred communication means to contact this buddy. In order to put the user in contact with his buddy, the Coordinator BPEL process has to retrieve information from the Database component, which is exposed in the framework as a web service (Database web service). The Coordinator BPEL process also needs to discover a suitable service in the Service Trader to provide the communication means selected by the user.

We realised the service trader as a UDDI registry using jUDDI [135], which is a Java implementation of the UDDI standard. Our jUDDI registry stores the descriptions of the services available in the framework. In our prototype, the available services are SMS, Phone, E-mail and Chat services. The service descriptions consist of XML documents with the name, type and endpoint of the service. The service type refers to semantic concepts described in an ontology supported by our framework. This ontology is based on the context model that we have defined at the PIM level of our methodology (see Section 6.1.2). The endpoint is the concrete address where the service is deployed. *Figure 110* shows an example of service description for the SMS service. To support the publication of service descriptions in this format, we have extended jUDDI with *tModels* that represent each of the service parameters, i.e., name, type and endpoint. To group the name, type and endpoint *tModels* under the same service, we have used the *categoryBag* UDDI element.



```
<ServiceDescription>
  <Name>SMS</Name>
  <Type>http://localhost:8080/ontologies/LiveContacts.owl#SmsService</Type>
  <Endpoint>http://localhost:8080/sms/services/sms</Endpoint>
</ServiceDescription>
```

Figure 110 SMS service description

Service descriptions are published in our jUDDI registry through the Publication web service depicted in *Figure 109*, which offers a publication interface to the service developers. This interface accepts a service description, parses this description and publishes the service name, type and endpoint in the jUDDI registry.

The Coordinator BPEL process can discover the services published in the jUDDI registry through the Discovery web service depicted in *Figure 109*. The discovery is based on the service type semantic concept used in the service descriptions. The discovery mechanism retrieves all the services with a type semantically related to the requested type. For example, assume that we are looking for the service type FixedPhoneService, which is a semantic concept as shown in the excerpt of the framework ontology depicted in *Figure 111*.

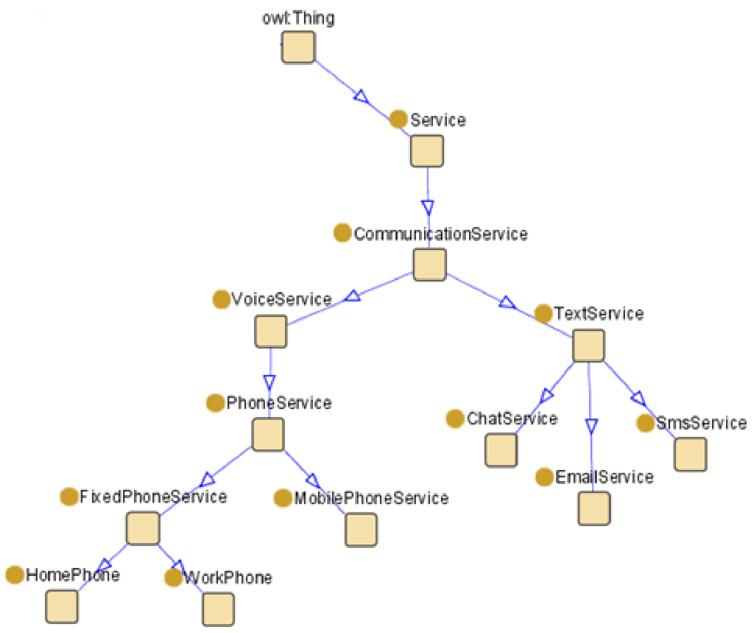


Figure 111 Framework ontology excerpt

The discovery mechanism retrieves the following matches, which are semantically related to the requested type:

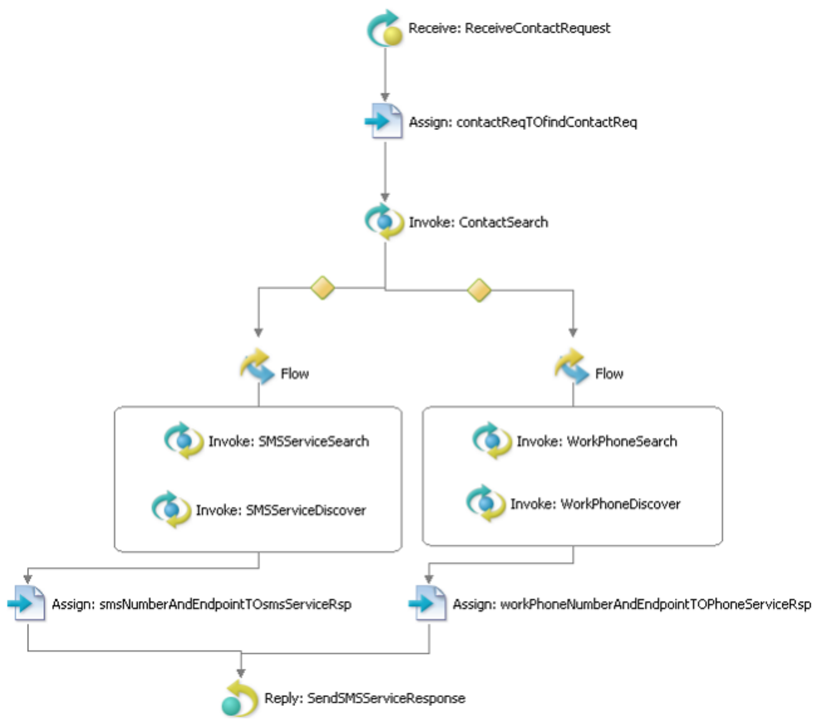
1. FixedPhoneService ⊂ PhoneService (FixedPhoneService is a *subsume* match of PhoneService);
2. FixedPhoneService ⊃ WorkPhone (FixedPhoneService is a *plug in* match of WorkPhone);
3. FixedPhoneService ⊃ HomePhone (FixedPhoneService is a *plug in* match of HomePhone);
4. FixedPhoneService ≡ FixedPhoneService (FixedPhoneService is an *exact* match of FixedPhoneService).

The discovery mechanism selects the *best match* among the options above. The best match is the *exact* match, followed by the *plug in* matches and then by the *subsume* match. The Discovery web service in *Figure 109* returns the endpoint of the best match to the Coordinator BPEL process. We realised the publication and discovery mechanisms as web services, so that they are eventually accessible from any component of the framework. The publication and discovery mechanisms are based on [136]. The Coordinator BPEL process finishes once the service endpoint has been discovered in the jUDDI registry and the contact details of the buddy have been retrieved from the Database. Endpoint and contact details are given as output (RequestOutputs) to the Coordinator client located in the User Agent. *Figure 109* shows that the User Agent also contains the clients to invoke the SMS, Phone, E-mail and Chat services (one client for each service). These are generic clients for the services, i.e., they do not have a specific service endpoint. Once it obtains the endpoint, the User Agent can finally invoke the proper communication service (service invocation) by giving the contact details of the buddy. This should put the user in contact with his buddy via the desired communication channel.

9.4 Implementation

Figure 112 shows the BPEL process that implements the coordinator, which orchestrates the components of our platform-specific framework. We manually designed this process by using the SDCM in *Figure 108* as source model. The automation of the transformation from PIM to PSM (transformation T_3 in *Figure 104*) is out of the scope of this thesis, but has been addressed in [124].

Figure 112 Platform-Specific design model (PSM): the coordinator BPEL process


Receive: ReceiveContactRequest
Assign: contactReqTOfindContactReq
Invoke: ContactSearch
Flow
Flow
Invoke: SMSServiceSearch
Invoke: SMSServiceDiscover
Invoke: WorkPhoneSearch
Invoke: WorkPhoneDiscover
Assign: smsNumberAndEndpointTOsmsServiceRsp
Assign: workPhoneNumberAndEndpointTOPhoneServiceRsp
Reply: SendSMSServiceResponse

The BPEL process in *Figure 112* starts with a *receive* activity with name ReceiveContactRequest that accepts as inputs the name of the buddy and the communication means to contact this buddy. The *assign* activity contactReqTOfindContactReq copies the name of the buddy contained in the contactReq message of the RemoveReceiveRequest activity to the findContactReq message attached to the *invoke* activity called ContactSearch. The invoke activity invokes the database web service in order to retrieve the information associated to the considered buddy. The BPEL process in *Figure 112* continues in two alternative flows, for either "SMS" or "WorkPhone" as selected communication means, respectively. We did not consider the other communication means in *Figure 112*, since their flow of activities is analogous to these two options. These flows execute two *invoke* activities in parallel named SmsServiceSearch and SmsServiceDiscover. The SmsServiceSearch activity invokes the database service to retrieve the contact details of the buddy, and the SmsServiceDiscover activity invokes the discovery web service to discover the endpoint of the service. When both *invoke* activities in the flow have been performed, their output is assigned to the *reply* activity with name SendSmsServiceResponse, which ends the BPEL process. The SendSmsServiceResponse activity sends the outputs of the process to the coordinator client in the user agent.

9.5 PIM to PSM Transformation

Figure 113 shows possible source and target models for our PIM to PSM transformation. The source model consists of the SDCM model in *Figure 108*, where only the internal details of the coordinator component and its collaboration with the other components are shown, since these details are relevant for the realisation of the coordinator BPEL process at the PSM level. The target model consists of the BPEL process shown in *Figure 112*. This PIM to PSM transformation could in principle be obtained in consecutive refinements, similarly to our refinements at the PIM level. However, in this thesis we assume that the PSM is obtained with only one transformation step.

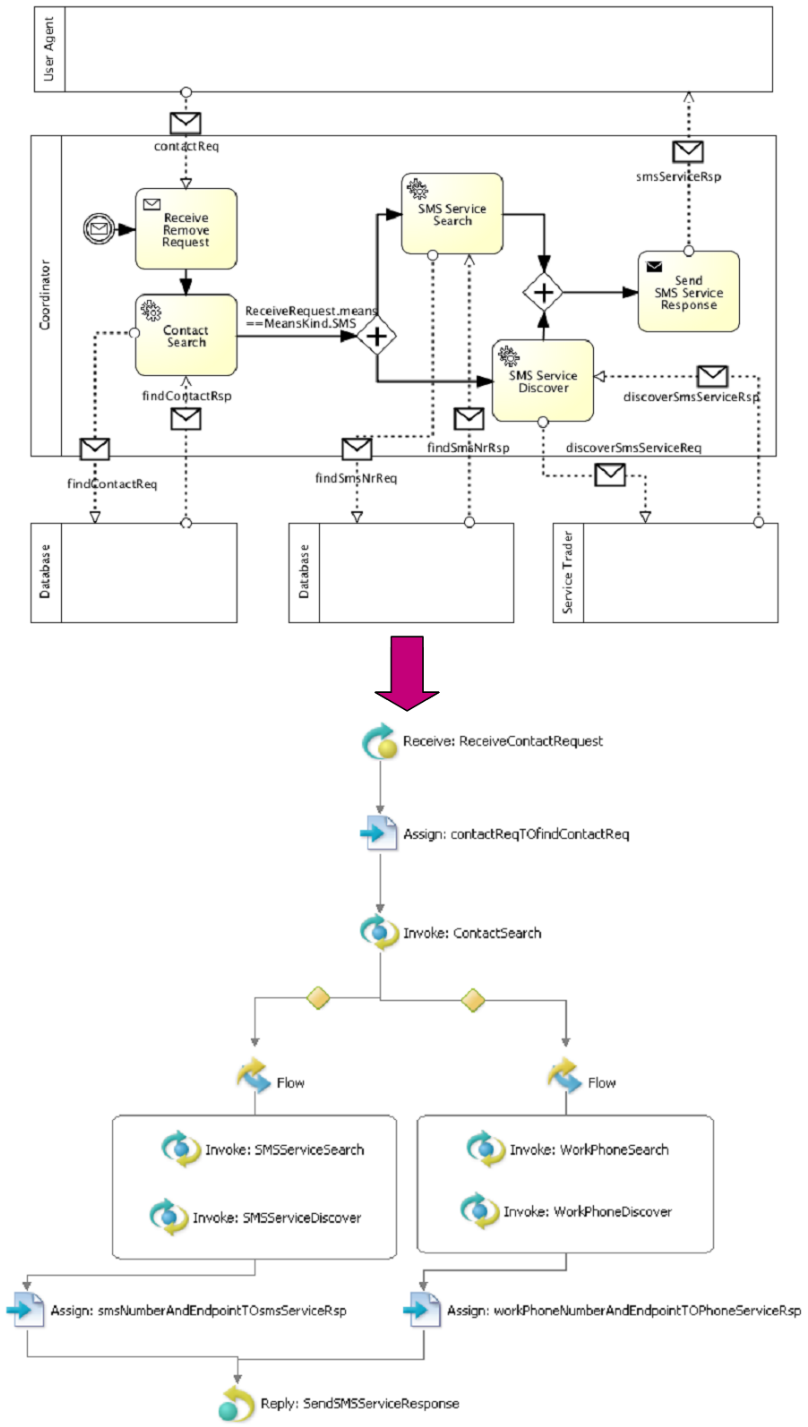


Figure 113 PIM to PSM transformation: source and target models

Although the complete automation of the PIM to PSM transformation in *Figure 113* is out of the scope of this thesis, we provide in *Figure 114* to *Figure 120* mappings that can be used to automate this transformation using SDCM executable patterns at the PIM level and implementation patterns at the PSM level (see Section 3.2).

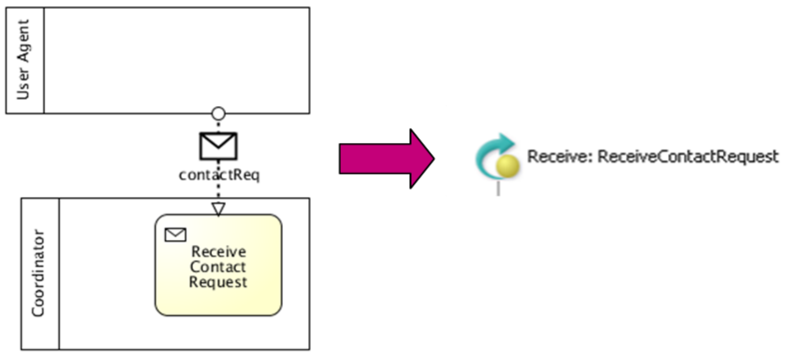


Figure 114 Contact Request pattern

Figure 114 shows the mapping of a one-way *request* executable pattern called Contact Request between the participants User Agent and Coordinator that exchange a contactReq message, onto a corresponding *receive* BPEL activity named ReceiveContactRequest and input message contactReq.

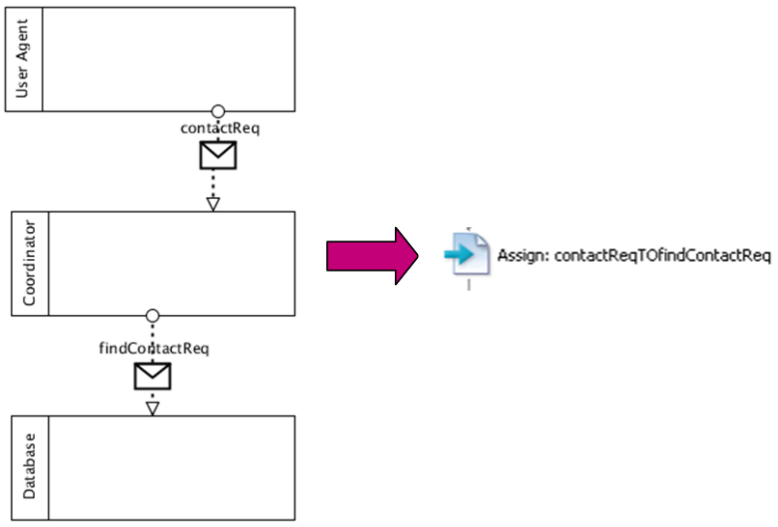


Figure 115 Assignment of *contactReq* input message to *findContactReq* output message

Figure 115 shows the assignment of the contactReq message of the ReceiveContactRequest *receive* activity shown in *Figure 114* as input value for the ContactSearch *invoke* activity discussed in the following mapping.

Figure 116 Contact Search pattern

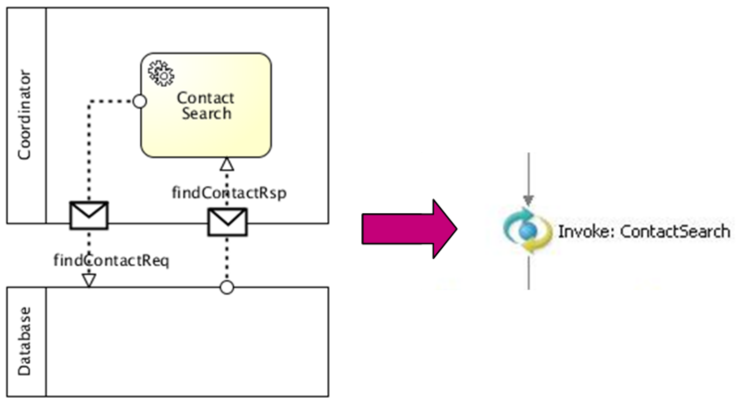


Figure 116 shows the mapping of a two-way *search* executable pattern called Contact Search between the participants Coordinator and Database that exchange the findContactReq and findContactRsp messages, onto a corresponding *invoke* BPEL activity named ContactSearch that sends a findContactReq request to the database and receives back a findContactRsp response.

Figure 117 SMS Service Search pattern

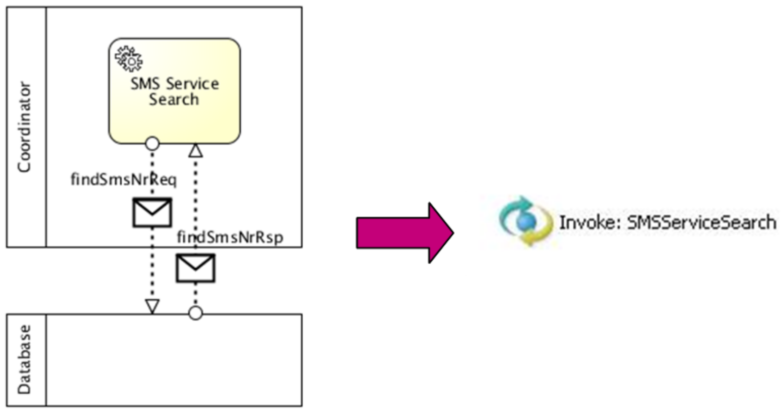


Figure 117 shows the mapping of a two-way *search* executable pattern called SMS Service Search that involves the participants Coordinator and Database, which exchange the findSmsNrReq and findSmsNRsp messages. The SMS

Service Search executable pattern is mapped onto a corresponding *invoke* BPEL activity named SMSServiceSearch that sends a findSmsNrReq request to the database and receives back a findSmsNRsp response.

Figure 118 SMS Service Discover pattern

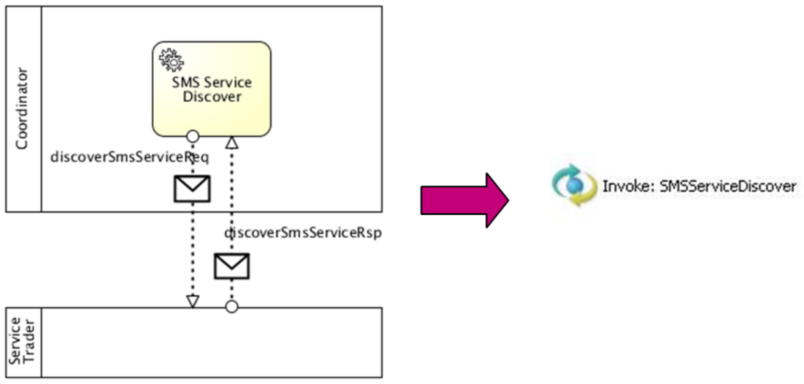


Figure 118 shows the mapping of a two-way *discover* executable pattern called SMS Service Search between the participants Coordinator and Service Trader that exchange the discoverSmsServiceReq and discoverSmsServiceRsp messages, onto a corresponding *invoke* BPEL activity named SMSServiceSearch that sends a discoverSmsServiceReq request to the service trader and receives back a discoverSmsServiceRsp response.

Figure 119 SMS Service Response pattern

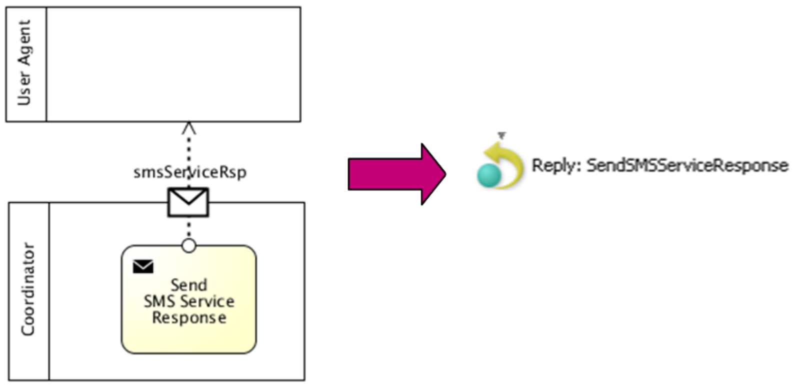


Figure 119 shows the mapping of a one-way *response* executable pattern called SMS Service Response between the participants Coordinator and User Agent that exchange an smsServiceRsp message, onto a corresponding *reply* BPEL activity named SendSMSServiceResponse and a variable smsServiceRsp.

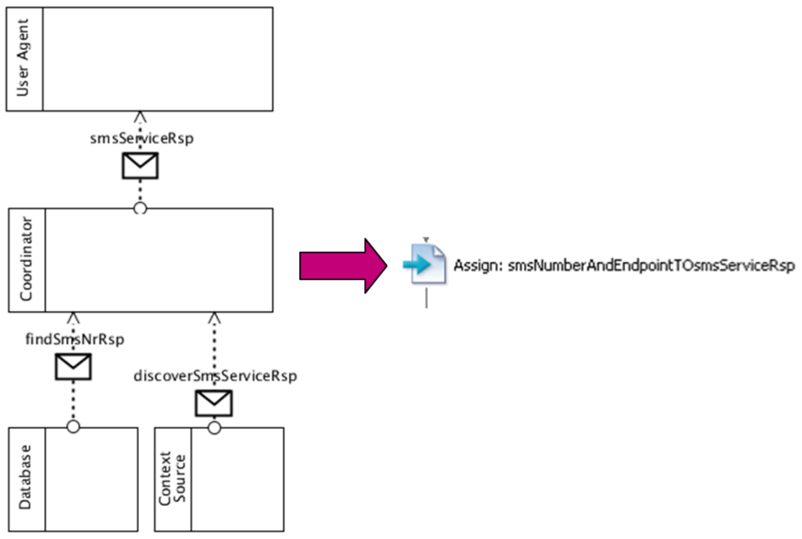


Figure 120 Assignment of input messages to *smsServiceRsp* output message

Figure 120 shows the assignment of the output of the SmsServiceSearch and SmsServiceDiscover *invoke* activities shown in *Figure 117* and *Figure 118*, respectively, to the input value of the SendSMSServiceResponse *reply* activity shown in *Figure 119*.

9.6 Discussion

In this chapter we have provided a case study that covers all the development steps prescribed by our methodology, including the realisation of a running prototype. This case study started with a service specification (SS) represented as a BPMN choreography diagram (see *Figure 105*), in which we identified *interaction markers* as placeholders for refined interactions at lower abstraction levels. This SS has been then given as input to our prototype *SStoSDRM refinement transformation* (see Section 8.3), which automatically generated a service design refined model (SDRM) represented as a more detailed BPMN choreography. This SDRM model is composed by *interaction patterns* that replaced the interaction markers used at the SS level to mark abstract interactions. The SDRM model has been given as input to a second transformation, namely our prototype *SDRMtoSDCM synthesis transformation* (see Section 8.5), which automatically generated a service design component model represented as a BPMN collaboration diagram. This SDCM model is composed by executable patterns that synthesized an internal behaviour for the corresponding interaction patterns used at the SDRM level. The SS, SDRM and SDCM models in BPMN can be alternatively modelled and transformed using A-MUSE DSL and ISDL, as we have done in Chapter 6, or using transition systems, as we have done in Chapter 7. Independently on the specific technique used to generate the SDCM model, we have used this model as input to realise a BPEL process for the coordinator component of our architecture. This BPEL process interacts with the other components of the reference architecture, which are realised as web services in our prototype, by means of the services that these components make available in a UDDI registry according to the SOA paradigm. The *PIMtoPSM transformation* from BPMN to BPEL process models translates a platform-independent model that can be in principle realised with any middleware target platform that supports service invocations, to a platform-specific model that depends on the specific technology chosen as target platform. Since this *PIMtoPSM transformation* is not the focus of this thesis, we have manually designed it for the proposed prototype for demonstration purposes. However, mappings from BPMN to BPEL are available in the literature [124].

The prototype presented in this chapter is one of the possible realisations of our service design component model with some target technologies. Other technologies than BPEL, UDDI and web services can be used. With this prototype, we showed that our methodology can be applied to generate running implementations. We did not consider here actual concrete context source components that retrieve context information from the user environment and generate context events in case of context changes. The integration of these components in our reference architecture using a context expression evaluator is discussed in [131]. However, we envision an integration of these components with web services technologies. By implementing the action providers as web services, we learned that this is a suitable solution to obtain flexibility, interoperability and portability in our platform-specific framework. Further study needs to be performed in order to integrate context source components in the framework and expose them as web services. These components require mechanisms to allow the coordinator to dynamically subscribe to context events as soon as these components become available to the application. However, we believe that our experiments with action providers as web services reported in this thesis have brought us a step forward towards the usage of context sources with this technology.

Chapter 10

Conclusions

This chapter presents the conclusions of this thesis and identifies topics that we recommend for future research. This chapter is further structured as follows: Section 10.1 presents some general remarks on our research, Sections 10.2 to 10.5 discuss the most important contributions of this work, and, finally, Section 10.6 presents directions for future research.

10.1 General Remarks

In today's market of service offerings, service providers not only have to create services that are innovative and distinctive in order to retain and attract demanding users, but also have to introduce these services quickly and effectively to remain competitive in their business. Therefore, a service development process that is agile, intuitively appealing to use, automated, and reusable has emerged as an important and desirable feature for service providers. In this thesis we have defined methodological support for such a development process.

We have provided a layered methodology based on behaviour modelling and transformations for the development of context-aware mobile applications, which are distributed applications that can provide innovative and distinctive services to their users. We have used currently available approaches, such as Service-Oriented Architecture (SOA) and Model-Driven Architecture (MDA), to support our methodology. SOA provided us with the architectural discipline necessary to define a reference architecture for context-aware mobile applications in which components interoperate using each other's services, abstracting from irrelevant implementation details. MDA provided us with the necessary design concepts and principles, such as, for example, the separation of platform-independent (PIM) and platform-specific models (PSM) concerns, the systematic (re)use of models, and the (automatic) use of model transformations. We have used these principles to progress the state-of-the-art in model-driven development of context-aware mobile applications by taking into account the behaviour of these applications already in early stages of the development process. In order to achieve this, in our methodology we have refined the application behaviour in several steps, from abstract specifications to final implementations. We have realised automated model transformations throughout these refinement steps to generate executable models and we have reasoned about their behavioural correctness.

The main contributions of this thesis can be summarised as follows:

– provide a layered methodology for behaviour modelling,
– promote proper communication between stakeholders,
– provide architectural support for context-aware mobile applications,
– develop automated support for behaviour model transformations.

The following sections discuss these contributions.

10.2 Layered Methodology for Behaviour Modelling

By developing the layered methodology for behaviour modelling proposed in this thesis, we have provided the following contributions.

Progress in state-of-the-art of model-driven development

According to model-driven development principles, in Chapters 2 and 3 we have prescribed the use of models as main artefacts of the development process and motivated the use of model transformations to refine the application behaviour from abstract specifications towards implementation. We argued that model-driven development has often focused on structural aspects, giving less attention to the behaviour of the application under development. This thesis contributed to the state-of-the-art in model-driven development by incorporating both the behaviour and the structure of the application under development in early stages of the development process, namely at the PIM level. In this way, the PIM level can be already used for behaviour analysis and simulation purposes, as opposed to the practise of dealing with these aspects at the end of the development process.

We also argued that the gap to be bridged by a PIM design is too big to be realised only in one step and more abstraction levels are necessary. A PIM design with only one abstraction level would bring either to a model with insufficient technical details for implementation, but understandable by humans, or to a model with all the necessary technical details, but hard to understand by humans. Therefore, we have divided the PIM design in several behavioural refinements that incrementally add technical details towards specific implementations by preserving the correctness of the original application behaviour. In order to demonstrate the novelty and general applicability of this contribution, in Chapter 4 we have presented a survey on techniques in model-driven behaviour modelling development that are relevant to this thesis, and classified these techniques according to our PIM behavioural refinements. We argued that none of these techniques can cover all these refinements. In Chapter 5 we have further compared these techniques according to qualitative criteria that are relevant in a methodology that aims at optimizing time, costs and efforts of the service development process. We do not claim that these qualitative criteria are ideal and complete, but they have been suitable for the purpose of our comparison. Based on this comparison, we have proposed three model-driven solutions that we used to implement our PIM behavioural refinement steps in Chapters 6, 7 and 8, respectively.

Language-independence

In Chapter 5, we claimed that our methodology is language-independent and can be used with different modelling languages as long as the chosen language allows us to properly model the behaviour of the application under development. This claim is justified in Chapters 6 to 8, in which we proposed three solutions that realise a running example employing different languages. The first solution (see Chapter 6) uses the A-MUSE domain-specific language and ISDL. The second solution (see Chapter 7) elaborates on an alternative that uses the formalism of Transition Systems. The third solution (see Chapter 8) proposes the usage of the BPMN notation. Although we used only this third solution in the case study presented in Chapter 9 due to space limitations, the other two solutions would also be suitable for this purpose.

Applicability

When designing an application, the separation of concerns principle promoted by the model-driven development prescribes that the application logic should be separated from the specific technology used to realise this application. Therefore, we have separated the design activities of our methodology in platform-independent design, which deals with application logic aspects, and platform-specific design, which is related to specific technological choices. While in Chapers 6 to 8 we have focused on the platform-independent design and its decomposition in behavioural refinements, in Chapter 9 we have applied these PIM behavioural refinements to generate a running prototype at the platform-specific level. The technology we have used at the PSM level consists of a BPEL process deployed in an BPEL execution engine, the web services invoked by this

BPEL process, and a jUDDI registry in which these services are registered and can be discovered by the BPEL process.

Behaviour correctness
We have defined behaviour model transformations that incrementally add technical details towards specific implementations. We have defined these model transformations to be correct by construction, i.e., we have created transformation rules that generate refined target models that preserve the original behaviour specified in the source model. In order to achieve this, we have initially designed by hand the source and target models for these transformations, and, afterwards, defined transformations rules between these models and automated them. When executing these automated transformations, we reasoned about the correctness of the generated target model by simply comparing it against the corresponding model specified by hand. However, we can extend our reasoning by using formalism to perform automated behaviour ananlysis of the generated models. Therefore, in Chapter 4 we investigated techniques for behaviour analysis that can be used to verify automatically behaviour correctness at different abstraction levels, and also the absence of undesirable behaviour, such as, for example, deadlocks, in the generated models. These techniques can also be used to assess the fulfillment of logical properties in order to validate the generated behaviour of an application against the user requirements. In Chapter 5, we have pointed out how some of these techniques can be integrated in our methodology. In Chapter 7, we have experimented with one of these techniques, which is based on transition systems formalisms.

10.3 Proper Communication between Stakeholders

We have used models as a means to promote common understanding between the stakeholders involved in the different phases of the development process. These models should be expressed in a language with a well-defined syntax and a formal semantics in order to avoid ambiguities and consequent misinterpretations. However, even with such models, proper communication between stakeholders cannot always be guaranteed due to their different background and skills. Every model has a purpose and models used by different stakeholders (can) have different purposes. For example, business experts would probably not understand a behavioural model expressed in some mathematical formalism or programming language, and, vice-versa, an IT developer would not be familiar with business models that describe how to create, deliver and capture value for an organization. Therefore, our contribution towards a proper communication between stakeholders consists of a (partially automated) methodology with different abstraction levels that allows each stakeholder to address the (same) development process at the most appropriate abstraction level, namely a higher abstraction level for business experts and a lower abstraction level for technical developers.

10.4 Architectural Support for Context-Aware Mobile Applications

An important challenge to be competitive in today's market of service offerings consists of providing the users with innovative, distinctive and user-centric services. Context-aware applications can offer services that are (1) *personalised* according to user preferences and history, (2) *ubiquitous* to be accessible wherever the users are and whatever activity they are doing, (3) *mobile* to serve users continuously connected to the Internet with their mobile devices, and (4) *composable* to facilitate the user experience. Therefore, we have tailored the methodology proposed in this thesis to context-aware mobile applications and provided the contributions discussed below to give architectural support to this family of applications.

SOA-based reference architecture

In Chapter 3, we have provided a reference architecture that supports general purpose functions used by context-aware applications. These general purpose functions are described in Chapter 2. For example, context source components are responsible for context gathering issues, such as collecting context conditions from sensors or web services and eventually aggregating these conditions in more complex context information. Action provider components are responsible for executing and delivering services as reactions to context changes or user input events. The coordinator component realises the application behaviour that controls context sources and action providers.

Since the components that constitute this reference architecture are distributed in the environment, we have used the SOA approach to facilitate their interoperability. In this way, components make use of each other's services to achieve the goals of the application without being concerned with the service implementation details. This is achieved in our reference architecture by exposing the descriptions of the services offered by the application components in a service registry. In Chapter 9, we have proposed a platform-specific framework in which the coordinator component is realised as a BPEL process. In this framework, action provider components are exposed as web services and registered to a UDDI registry in terms of their interface descriptions.

Interaction patterns connected to the reference architecture
We have identified basic patterns of behaviour and used them as building blocks to realise transformations between models at different abstraction levels. These patterns represent interactions that are performed by components of our reference architecture. In general, these patterns can be used to speed up and facilitate the development of new context-aware mobile applications based on this reference architecture, instead of building them from scratch, which tends to be costly and time-consuming.

10.5 Automated Support for PIM Behaviour Model Transformations

An important factor to speed up the development process of a software application is to automate development tasks, possibly along the overall development process. In this thesis, we addressed the automation of model transformations at the PIM level of our methodology. In order to achieve this, we have realised automatic model transformations, and also organised the knowledge acquired in the process of automating these transformations in order to make it available for reuse. Our contributions with regard to automation and reuse aspects are briefly discussed below.

Automation
We have implemented the *SStoSDRM behaviour refinement* and the *SDRMtoSDCM behaviour synthesis* transformations described in Chapters 6 to 8. These are transformations from a source model at a certain abstraction level to a (more concrete) target model at a lower abstraction level. In Chapter 6, we have implemented the *SStoSDRM behaviour refinement* using the Medini QVT engine, which allowed us to define transformation rules in the QVT Relation language defined by OMG. In Chapter 8, we have implemented the *SStoSDRM behaviour refinement* and *SDRMtoSDCM behaviour synthesis* using the ATL transformation language. Both the QVT and ATL engines allowed us to execute our transformations in an Eclipse-based environment. The emphasis of this thesis is not on pros and cons of transformation languages and tools, so that in Chapters 6 and 8 we have abstracted from the specific transformation languages chosen for implementing the transformations, and we have shown the mappings of source metamodel elements onto target metamodel elements only schematically. However, we have learned that both ATL and QVT are suitable languages for implementing refinement and synthesis transformations like the ones presented in this thesis.

Reusability
The design knowledge acquired during the development of our methodology is reusable and may be consolidated in terms of (1) artefacts that can be straightforwardly reused when applying our methodology to the development of similar applications, and (2) best practices that can be reused to guide the definition of new development methodologies.

In Chapters 6 to 8 we have discussed behavioural patterns that are reusable artefacts to be used as building blocks for creating complex behaviours. We have also provided transformation rules based on these patterns for automatic generation of executable behaviours from abstract specifications. Both behavioural patterns and transformation rules are tailored to a specific family of applications, namely context-aware mobile applications, and can be directly reused for developing new applications in this domain, instead of building them from scratch.

Concerning best practices, this thesis provides designers with guidelines to define new methodologies based on principles such as separation of concerns, systematic decomposition in abstraction levels, use of recurring patterns, definition of model tranformations between different abstraction levels, and the automation of these transformations.

10.6 Future work

We suggest the following directions for further research.

Case study development
We have identified the interaction patterns and developed the model transformations provided in this work using the Live Contacts running example. Starting with the results achieved with this running example, we have generalised the interaction patterns and transformation rules in order to create abstractions that can be used with other applications. However, we did not applied our results to other applications. As part of future work, the development of a case study based on a different context-aware mobile application should be considered, possibly also in another application area. Candidate areas could be, for example, health-care, government or domotics. We foresee that the development of a new application using the same reference architecture, interaction patterns and automated transformations developed in this thesis should be beneficial in terms of reduced development time, effort and costs compared to the development time, effort and costs necessary to create a new application from scratch.

Full automation

Although an important challenge in model-driven development consists of achieving automation along the overall development process, we acknowledge that full automation is probably impossible to achieve, at least with currently available technology. Full automation would require full composability, and full composability remains a difficult goal to achieve. In this thesis, we have provided partial automatic support for the proposed methodology, namely for the model transformations at the PIM level. However, we did not address the automation of the transformation(s) from PIM to PSM. Therefore, our work could be extended by providing automated support also for this transformation, for example, using the results from [111, 124], as initially investigated in Chapter 5. In this way, the entire chain of transformation steps prescribed by our methodology would be automated, bringing us a step forward towards the dream of realising full automation.

Behaviour correctness

In Chapter 8, we have realised transformations that automatically refine BPMN behavioural models at different abstraction levels, guaranteeing correctness with the original application behaviour by construction. However, we did not exploit behaviour correctness at a rigorous formal level. The natural evolution of the work proposed in Chapter 8 should be the integration of our methodology with techniques to formally prove the equivalence of our BPMN models to formalisms that can be automatically analise, such as, for example, Petri Nets. This could be done by applying the results of [97-98, 107], as initially investigated in Chapter 5.

Interaction pattern abstractions

In our work, a basic interaction pattern represents an interaction that involves two participants, which should be chosen among the set of components of our SOA-based reference architecture, namely context sources, user agent, coordinator, database, service trader, and action providers. Therefore, interaction patterns are connected to the specific reference architecture for context-aware mobile applications used in this thesis. Further investigation should aim at generalising these interaction patterns and apply minor adjustments in order to make them reusable also with other references architectures.

Context manager component

The control component of the SOA-based reference architecture proposed in this thesis consists of the *coordinator* component, which receives *context events* and triggers *actions* to be executed as a consequence. Context events, which consist of relevant changes in the user's environment, are provided to the coordinator by dedicated *context sources* components. To support interoperability between coordinator and context sources, we have defined a *context model*, which consists of a conceptual model that represents context information abstracting from any design and technological detail, such as the way this context is sensed, provided, learned, produced, or used. Our context model has been defined based on the Live Contacts running example, and, therefore, includes some application-specific concepts. However, this context model also includes a general part that can be reused for various context-aware mobile applications. We have addressed this interoperability aspect using a *context expression evaluator* component in our previous work [131]. As part of future work, we recommend that the design and implementation of a *context manager component* based on our context model should be considered to allow interoperability between the coordinator and context sources. To achieve this, the context manager component should be able to receive context events subscriptions from the coordinator, gather context information related to these subscriptions from the proper context source, reason about this information, and, finally, send events notifications to the coordinator.

References

1. Rehemtulla, M. and G. Hughes, *Service Creation: Meeting the Product Lifecycle Challenge* in *Product lifecycle management*. 2006, IBM.

2. Russo, P., *A Solid Foundation: an Integrated Architecture for Service Creation Can Help Carriers Speed New Services to Market*, in *Connected Planet Online*. 2009.

3. Lodge, F. (1998), *Service creation*. Lecture Notes in Computer Science **1430**, 363-364.

4. *Ubiquitous - Dictionary.com Page*. Available from: http://dictionary.reference .com/browse/ubiquitous.

5. *Pervasive - Dictionary.com Page*. Available from: http://dictionary.reference. com/browse/pervasive.

6. Naughton, B., *Standardizing the Process of Service Creation and Delivery for Telcos.* www.pipelinepub.com, 2006. **3**(12).

7. Watson, B., *Service Creation Best Practices*, in *Stratecast*. 2007, IBM.

8. Dobing, B. and J. Parsons (2005), *Current Practices in the Use of UML*. Lecture Notes in Computer Science **3770**, 2-11.

9. ObjectManagementGroup, *MDA-Guide*, *Version 1.0.1*. 2003: omg/03-06-01.

10. Papazoglou, M., *Service Oriented Computing.* Communications of the ACM, 2003. **46**(10): p. 25-28.

11. McNeile, A. and N. Simons, *Methods of Behaviour Modelling: A Commentary on Behaviour Modelling Techniques for MDA*, Metamaxim Ltd Home.

12. Cernosek, G., *The Value of Modeling*, in *Rational Software*, IBM.

13. Jochen, L., *Models in Software Engineering: an Introduction.* Software and Systems Modeling 2003. **2**(1): p. 5-14.

14. Scacchi, W., *Process Models in Software Engineering*, in *Encyclopedia of Software Engineering*. 2002.

15. Metzger, A., *A Systematic Look at Model Transformations*, in *Model-Driven Software Development*. 2005, Springer-Verlag. p. 19-33.

16. Bézivin, J., *In Search of a Basic Principle for Model Driven Engineering.* UPGRADE, 2004. **2**: p. 21-24.

17. Bézivin J. et al., *Model Engineering for Complex Systems.*

18. France, R. and Rumpe B., *Model-driven Development of Complex Software: A Research Roadmap.*

19. Shaham-Gafny, Y. and S. Kremer-Davidson, *MDD Enablement Tooling*. 2004, RAD Technologies.

20. *Model-Driven Architecture (MDA) Home*. Available from: http://www.omg.org/mda/.

21. *Object Managment Group Home*. Available from: http://www.omg.org/.

22. Object Management Group. *Unified Modelling Language (UML) specification, version 2.3* 2010; Available from: http://www.omg.org/spec/UML/.

23. Object Management Group. *XML Metadata Interchange (XMI) Specification* Available from: http://www.omg.org/spec/XMI/.

24. Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT)*. 2011; Available from: http://www.omg.org/spec/QVT/1.1.

25. Eclipse Foundation. *The ATLAS Transformation Language*. Available from: http://www.eclipse.org/atl/.

26. *Eclipse Modelling Framework (EMF) Project*. Available from: http://www.eclipse.org/modeling/emf/.

27. Gavreas, A., et al. (2004) *Towards an MDA-based Development Methodology*. Lecture Notes in Computer Science, 230-240.

28. Cabot, J. *Relationship between MDA, MDD and MDE*. MOdeling LAnguages: The portal for software modelers, analysts, designers and architects (2009).

29. den Haan, J. *MDA MDD MDE MDSD MDSE: help!* The Enterprise Architect: Building an Agile Enterprise (2008). Available from: http://www.theenterprisearchitect.eu/archive/2008/06/11/mda-mdd-mde-mdsd-mdse-help.

30. Favre, J.M. *Towards a Basic Theory to Model Driven Engineering*. in *The 3rd Workshop in Software Model Engineering*. 2004.

31. Andrade Almeida, J.P., *Model-Driven Design of Distributed Applications*. 2006, University of Twente: Enschede, The Netherlands.

32. Object Management Group, *CORBA Specification, version 3.1*. 2008; Available from: http://www.omg.org/spec/CORBA/.

33. W3C. *World Wide Web Consortium: Web Services Architecture*. 2004; Available from: http://www.w3.org/TR/ws-arch/.

34. Object Management Group, *Meta Object Facility (MOF) Core.* 2006.

35. Harel, D. and B. Rumpe, *Modeling Languages: Syntax, Semantics and All That Stuff (or, What's the SEmantics of "Semantics"?).* Computer 2004. **37**(10): p. 64-72.

36. Mens, T. and P. van Gorp. *A Taxonomy of Model Transformations*. in *The International Workshop on Graph and Model Transformation (GraMoT)* 2006: Elsevier Science.

37. Northrop, C., *Software Product Lines: Practices and Patterns*. 2001: Addison-Wesley Longman Publishing Co.

38. Brown, A.W. Rational Edge (2004); Available from: http://www.ibm.com/developerworks/rational/library/3100.html.

39. IKV++Technologies. *MediniQVT*. Available from: http://www.ikv.de.

40. Henderson-Sellers, B., *UML - the Good, the Bad or the Ugly? Perspectives from a Panel of Experts.* International Journal of Software and Systems Modelling (SoSyM) 2005. **4**(1): p. 4-13.

41. Papazoglou Mike , et al., *Service Oriented Computing Research Roadmap.* 2006.

42. Thomas, E., et al. *SOA manifesto*. Available from: http://www.soa-manifesto.org/.

43. Erl, T., *SOA: Principles of Service Design*. Service-Oriented Computing. 2007: Prentice Hall.

44. *Java Remote Method Invocation (Java RMI) home*. Available from: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

45. *Jini Home*. Available from: http://www.jini.org/wiki/Main_Page.

46. O'Sullivan Justin, e.a., *Service Description: A survey of the general nature of services.* 2002.

47. Papazoglou, M. and W.-J. van Den Heuvel, *Service Oriented Design and Development Methodology.* International Journal of Web Engineering and Technology (IJWET), 2006. **2**(4).

48. Ferreira Pires, L., *Architectural Notes: a Framework for Distributed Systems Development*, Ph.D. Thesis, University of Twente, the Netherlands, 1994.

49. *Merriam-Webster Online*. Available from: http://m-c.com.

50. Dockhorn Costa, P., *Architectural Support for Context-Aware Applications: from Context Models to Services Platforms*, Ph.D. Thesis, University of Twente, the Netherlands, 2007.

51. Dey, A.K., G.D. Abowd, and D. Salber, *A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications.* Human-Computer Interaction, 2001. **16**: p. 97-166.

52. A.K. Dey, G.D.A., D. Salber, *A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications.* Human-Computer Interaction, 2001. **16**: p. 97-166.

53. K. Henricksen, J.I. (2004) *A software engineering framework for context-aware pervasive computing*. Proceedings of the 2th Conference on Pervasive Computing and Communications, 77-86.

54. A. Ranganathan, R.H.C. (2003) *A Middleware for context aware agents in ubiquitous computing environments*. Lecture Notes in Computer Science **2672**, 143-161.

55. Henricksen, K. and J. Indulska (2004) *A software engineering framework for context-aware pervasive computing*. Proceedings of the 2th Conference on Oervasive Computing and Communications , 77-86.

56. Dockhorn Costa, P., L. Ferreira Pires, and M. van Sinderen. *Architectural PAtterns for Context-Aware Applications*. in *The 2nd International Workshop on Ubiquitous Computing (IWUC)*, 2005.

57. Guizzardi, G., *Ontological Foundations for Structural Conceptual Models*, Ph.D. Thesis, University of Twente, the Netherlands, 2005.

58. Gavreas A., B.M., Ferreira Pires L., Almeida J.P. (2004) *Towards an MDA-based Development Methodology*. Lecture Notes in Computer Science, 230-240.

59. Ferreira Pires L., Q.D., van Sinderen M., Vissers C., *Architecture of Distributed Systems, Lecture notes*. 2007, University of Twente.

60. *Live Contacts Home*. Available from: http://livecontacts.telin.nl.

61. *A-MUSE Project Home*. Available from: http://www.freeband.nl/project.cfm?language=en&id=489.

62. Object Management Group. *Trading Object Services Specification, version 1.0*. 2000; Available from: http://www.omg.org/spec/TRADE/.

63. OASIS. *Universal Description Discovery and Integration (UDDI) project*. Available from: http://uddi.xml.org/specification.

64. Uchitel, S., G. Brunet, and M. Chechick. *Behaviour Model Synthesis from Properties and Scenarios* in *29th International Conference on Software Engineering (ICSE)*. 2007. Minneapolis, USA: IEEE Computer Society Press.

65. Giannakopoulou, D. and J. Magee. *Fluent Model Checking for Event-Based Systems*. in *9th European Software Engineering Conference (ESEC 2003) and the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2003)*. 2003. Finland: ACM Press.

66. ITU. *Recommendation z.120: Message Sequence Charts*. 2000.

67. Uchitel, S., J. Kramer, and J. Magee, eds. *Incremental Elaboration of Scenario-Based Specifications and Behaviour Models Using Implied Scenarios*. ACM Transactions on Software Engineering and Methodology. Vol. 13. 2004, ACM Press. 37-85.

68. Imperial College London and University of Buenos Aires, *Modal Transition Systems Analyser (MTSA)*.

69. Imperial College London, *Labelled Transition Systems Analyser (LTSA)*.

70. Foster, H., *A Rigorous Approach to Engineering Web Service Compositions*, Ph.D. Thesis, Imperial College London, 2006.

71. Harel, D., *Can Programming Be Liberated, Period?*, in *IEEE Computer*. 2008, IEEE Computer Society. p. 28-37.

72. Harel, D. and R. Marelli, *Specifying and Executing Behavioral Requirements: The Play-In/ Play-Out Approach*, in *Tecnical Report*. 2001, The Weizmann Institute of Science.

73. Brill, M., et al. *Live Sequence Charts*. in *Integration of Software Specification Techniques for Applications in Engineering*. 2004: Springer.

74. Harel, D., et al. *PlayGo: Towards a Comprehensive Tool for Scenario Based Programming*. in *International Conference on Automated Software Engineering (ASE)*. 2010. Antwerp, Belgium: ACM.

75. *PlayGo Home*. Available from: www.playgo.co.

76. *AspectJ Home*. Available from: http://www.eclipse.org/aspectj/.

77. Harel, D. and H. Kugler, *Synthesizing State-Based Object Systems from LSC Specifications.* International Journal of Foundations of Computer Science, 2002. **13**(1): p. 5-51.

78. *Rhapsody Home*. Available from: http://www-01.ibm.com/software/ awdtools/rhapsody.

79. *Fujaba Tool Suite Home*. Available from: http://www.fujaba.de.

80. Fischer, T., et al. (2000) *Story Diagrams: a New Graph Rewrite Language Based on the Unified Modeling Language and Java*. Lecture Notes in Computer Science **1764**, 296-309.

81. Rozenberg, G., *Handbook of Graph Grammars and Computing by Graph Transformations. Foundations.* Vol. 1. 1997: World Scientific Publishing Co.

82. Engels, G., et al. *From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations* in *The 4th European Conference on Model Driven*

Architecture - Foundations and Applications (ECMDA-FA). 2008. Lecture Notes in Computer Science.

83. Kastenberg, H., A. Kleppe, and A. Rensink. *Defining Object-Oriented Execution Semantics using Graph Transformations*. in *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. 2006: Springer-Verlag.

84. Rensink, A. *The GROOVE Simulator: a Tool for State Space Generation* in *Fifth Annual International Working Conference on Active Networks (AGTIVE 2003)*. 2003: Springer

85. Raedts, I., et al. *A Software Framework for automated Verification*. in *22nd Annual Symposium on Applied Computing*. 2007.

86. Raedts, I., et al. *Transformation of BPMN Models for Behaviour Analysis*. in *5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*. 2007: INSTICC Press.

87. van Hee, K., R. Post, and L. Somers. *Yet Another Smart Process EditoR*. in *European Simulation and Modelling Conference*. 2005.

88. W3C. *XSL Transformations (XSLT) Version 2.0*. 2007; Available from: http://www.w3.org/TR/xslt20/.

89. Verbeek, H.M.W., T. Basten, and W.M.P. van der Aalst, *Diagnosing Workflow Processes using Woflan.* The computer Journal, 2001. **44**(4): p. 246 - 279.

90. Roch, S. and P. Starke, *INA: I Integrieter Netzanalysator*, Humboldt Universitaet zu Berlin.

91. Schmidt, K. *LoLA: a Low Level Analyser*. in *International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN)*. 2000: Springer-Verlag.

92. Groote, J.F., et al., *The Formal Specification Language mCRL2*, in *Methods for Modelling Software Systems*. 2007, Internationales Begegnungs- und Forschungszentrum Informatik.

93. *mCRL2 Home*. Available from: http://www.mcrl2.org.

94. Object Management Group. *BPMN Home*. Available from: http://www.omg.org/spec/BPMN.

95. Object Management Group. *BPMN 1.0 Specification Home*. Available from: http://www.bpmn.org/Documents/OMG_Final_Adopted_BPMN_1-0_Spec_06-02-01.pdf.

96. Dijkman, R.M. *BPMN*. Available from: http://is.ieis.tue.nl/staff/rdijkman/Homepage_Remco_Dijkman/BPMN.html.

97. Dijkman, R.M., M. Dumas, and C. Ouyang, *Semantics and Analysis of Business Process Models in BPMN.* Information and Software Technology (IST), 2008. **50**(12): p. 1281-1294.

98. Dijkman, R.M. and P. Van Gorp. *BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules*. in *International Workshop on BPMN*. 2010.

99. van der Aalst, W.M.P. and A.H.M. Hofstede, *YAWL: Yet Another Workflow Language.* Information Systems, 2004. **30**(4): p. 245 - 275.

100. Billington, J., s. Christensen, and K. van de Hee, *The Petri Net Markup Language: Concepts, Technology and Tools*, in *Applications and Theory of Petri Nets*. 2003, Springer. p. 483 - 505.

101. van Dongen, B., et al., *The ProM Framework: a New Era in Process Mining Tool Support* in *Application and Theory of Petri Nets*. 2005, Springer. p. 444 - 454.

102. WFMC, *Workflow Management Coalition Standard: Workflow Process Definition Interface - XML Process Definition Language (XPDL)*, in *Technical Report*. 2002, Workflow Management Coalition.

103. Jakumeit, E., S. Buchwald, and M. Kroll, *GrGen.NET.* International Journal on Software Tools for Technology Transfer (sTTT), 2010.

104. Ouyang, C., et al., *Pattern-based Translation of BPMN Process Models to BPEL Web Services.* International Journal of Web Services Research (JWSR), 2007. **5**(1): p. 42-62.

105. Jordan, D. and J. Evdemon. *WebServices Business Process Execution Language (WS-BPEL) Specification Version 2.0*. 2007.

106. Mendling, J., K.B. Lassen, and U. Zdun, *On the Transformation of Control Flow between Block-Oriented and Graph-Oriented Process Modeling Languages.* International Journal of Business Process Integration and Management (IJBPIM). Special Issue on Model-Driven Engineering of Executable Business Process Models, 2008. **3**(2): p. 96-108.

107. Lohmann, N., et al., *Analyzing interacting WS-BPEL processes using flexible model generation.* Data and Knowledge Engineering, 2008. **64**(1): p. 38-54.

108. Massuthe, P. and K. Schmidt, *An Operating Guideline Approach to the SOA.* Annals of Mathematics, Computing and Teleinformatics, 2005. **1**(3): p. 35 - 43.

109. *Tools4BPEL Home*. Available from: http://www2.informatik.hu-berlin.de/top/tools4bpel.

110. *Fiona Home*. Available from: http://www2.informatik.hu-berlin.de/top/tools4bpel/fiona/.

111. Dirgahayu, T., *Interaction Design in Service Compositions*, Ph.D. Thesis, University of Twente, The Netherlands, 2010.

112. Quartel, D., *Action Relations. Basic Design Concepts for Behaviour Modelling and Refinement*, Ph.D. Thesis, University of Twente, The Netherlands, 1998.

113. *ISDL Home*. Available from: http://isdl.ctit.utwente.nl.

114. Combes, P., D. Harel, and H. Kugler, *Modeling and Verification of a Telecommunication Application Using Live Sequence Charts and the Play-Engine Tool.* Software and Systems Modeling, 2008. **7**(2): p. 157-175.

115. Bontemps, Y., *Automated Verification of State-Bases Specification against Scenarios - a Srep Towards relating Inter-Obect to Intra-Object Specifications*. 2001, Facultés Universitaires Notre-Dame de la Paix: Namur, Belgium.

116. Topçu, O., M. Adak, and H. Oguztüzün, *Metamodeling Live Sequence Charts for Code Generation.* Software and Systems Modeling, 2009. **8**(4): p. 567-583.

117. *PNML.org Home*. Available from: http://www.pnml.org/grammar.php.

118. Uchitel, S. and J. Kramer. *A Workbench for Synthesising Behaviour Models from Scenarios*. in *23rd IEEE/ACM International Conference on Software Engineering (ICSE)*. 2001: ACM.

119. Klose, J., *Live Sequence Charts: a Graphical Formalisms for the Specification of Communication Behavior*. 2003.

120. Kugler, H., et al. *Temporal Logic for Scenario-Based Specifications, Proc. of the*

in *The 11th Inter. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* 2005: Springer.

121. Kumar, R., E. Mercer, and A. Bunker. *Improving Translation of Live Sequence Charts to Temporal Logic*. in *The 7th International Conference on Automated Verification of Critical Systems (AVoCS)* 2007.

122. Brill, M., et al. *Formal Verification of LSCs in the Development Process*. in *Integration of Software Specification Techniques for Applications in Engineering*. 2004: Springer.

123. Hausmann, J.H., *Dynamic Meta Modelling*, Ph.D. Thesis, University of Paderborn, Germany, 2005.

124. Ouyang, C., et al., *Pattern-based Translation of BPMN Process Models to BPEL Web Services.* International Journal of Web Services Research (JWSR), 2008. **5**(1): p. 42-62.

125. Quartel, D. *Simulation and Execution of Service Models Using ISDL*. in *ACT4SOC*. 2008: INSTICC Press.

126. Heerink, L. and D. Quartel, *Domain Specific Language for Context-Aware Mobile Services*, in *A-MUSE Project Deliverable*. 2007.

127. Daniele, L., L. Ferreira Pires, and M. van Sinderen. *An MDA-Based Approach for Behaviour Modelling of Context-Aware Mobile Applications*. in *The Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA)*. Lecture Notes in Computer Science, 2009.

128. Daniele, L., L. Ferreira Pires, and M. van Sinderen. *Towards Automatic Behavior Synthesis of a Coordinator Component for Context-Aware Mobile Applications*. in *The 13th Enterprise Distributed Object Computing Conference Workshops (EDOCW)*. 2009: IEEE Computer Society Press.

129. *Business4Users (B4U) Project Home*. Available from: http://www.freeband.nl/kennisimpuls/projecten/b4u/ENindex.html.

130. Ter Hofte, G.H., et al. (2004) *Context-Aware Communication with Live Contacts*. Conference Supplement of Computer Supported Cooperative Work (CSCW 2004).

131. Daniele, L.M., L. Ferreira Pires, and M. van Sinderen. *Context Handling in a SOA Infrastructure for Context-Aware Applications*. in *The 2nd International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing (ACT4SOC 2008)*. 2008. INSTICC Press.

132. *Labelled Transition Systems Analyser (LTSA) Home*. Available from: http://www.doc.ic.ac.uk/ltsa.

133. *Modal Transition Systems Analyser (MTSA) Home*. Available from: http://sourceforge.net/projects/mtsa/.

134. *Signavio-Oryx Initiative Home*. Available from: http://www.signavio.com/en/academic.html.

135. *jUDDI Home*. Available from: http://ws.apache.org/juddi.

136. Goncalves da Silva, E.M., L. Ferreira Pires, and M. van Sinderen, *Towards Runtime Discovery, Selection and Composition of Semantic Services.* Computer communications, 2011. **34**(2): p. 159 -168.

About the author

Laura Maria Daniele was born in Cagliari, Italy, on the 20th of July 1980. She obtained her high school diploma in the humanistic area (History, Philosophy, Italian, Latin, Greek grammar and literature). Afterwards, she moved to the technical area and studied Electronics at the Department of Electrical and Electronic Engineering at the University of Cagliari. She graduated in 2006 and close to the conclusion of her studies, she spent seven months at the Architecture and Services of Network Applications (ASNA) group at the University of Twente by means of the Socrates/Erasmus exchange program. During this time she developed her master's thesis work in the context of the Freeband AWARENESS project. In February 2007 she started her PhD research at the Centre of Telematics and Information Technology (CTIT), University of Twente, as a member of the Software Engineering group (also known as Twente Research & Education on Software Engineering, TRESE). During her PhD work, she was involved in the Freeband A-MUSE project. Her professional interests include model-driven design methodologies, behaviour modelling techniques, model transformations, service-oriented architectures and context-aware mobile applications. She has authored several international publications and served as a reviewer for international conferences and workshops.

Below is a list of her publications in reverse chronological order:

– Daniele, L.M., Ferreira Pires, L. and van Sinderen, M.J. (2010): *Process-Oriented Behavior Generation Using Interaction Patterns*. In: 14th IEEE International Enterprise Distributed Object Computing Conference

Workshops (EDOCW 2010), 25-29 Oct 2010, Vitoria, Brazil. pp. 15-20. IEEE Computer Society.

- Daniele, L.M., Ferreira Pires, L. and van Sinderen, M.J. (2009): *An MDA-based approach for behaviour modelling of context-aware mobile applications*. In: Proceedings of the Fifth European Conference on Model-Driven Architecture Foundations and Applications, ECMDA-FA 2009, 23-26 June 2009, Enschede, The Netherlands. pp. 206-220. Lecture Notes in Computer Science 5562. Springer Verlag.

- Daniele, L.M., Ferreira Pires, L. and van Sinderen, M.J. (2009): *Towards automatic behavior synthesis of a coordinator component for context-aware mobile applications*. In: 13th Enterprise Distributed Object Computing Conference Workshops, EDOCW 2009, 1-4 Sept 2009, Auckland, New Zealand. pp. 140-147. IEEE Computer Society.

- Daniele, L.M., Goncalves da Silva, E.M., Ferreira Pires, L. and van Sinderen, M.J. (2009): *A SOA-based platform-specific framework for context-aware mobile applications*. In: Proceedings of the Second IFIP WG5.8 Workshop on Enterprise Interoperability, IWEI 2009, 13-14 Oct 2009, Valencia, Spain. pp. 25-37. Lecture Notes in Business Information Processing 38. Springer Verlag.

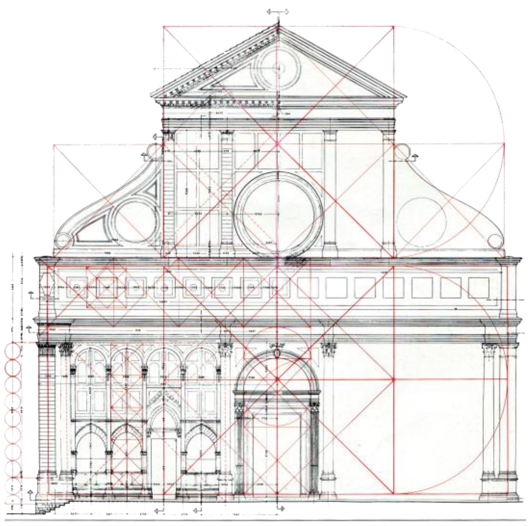
- Daniele, L.M., Ferreira Pires, L. and van Sinderen, M.J. (2008): *Interaction patterns for refining behaviour specifications of context-aware mobile services*. In: Joint Proceedings of the Workshops: IWUC, MDIES and TCoB, 12 Jun 2008, Barcelona, Spain. pp. 64-76. INSTICC Press.

- Daniele, L.M., Ferreira Pires, L. and van Sinderen, M.J. (2008): *Context handling in a SOA infrastructure for mobile applications*. In: Proceedings of the 2nd International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing (ACT4SOC 2008), 5 July 2008, Porto, Portugal. pp. 27-37. INSTICC Press.

- Daniele, L.M., Dockhorn Costa, P. and Ferreira Pires, L. (2007): *Towards a rule-based approach for context-aware applications*. In: Proceedings of the 13th Open European Summer School and IFIP TC6.6 Workshop, EUNICE 2007, July 18-20, 2007, Enschede, The Netherlands. pp. 33-43. Lecture Notes in Computer Science 4606. Springer Verlag.

About the cover

This thesis discusses how to build software applications in a systematic way using a methodology that divides the design phase, which is based on the architecture of these applications, from the implementation phase, in which this architecture is realised using concrete components of the real world. Similarly, in construction engineering the realisation of a building is preceded by a careful and proper design phase. Since I come from Italy, which is a country with centuries of history and art, I thought it would be nice to have in my cover an example of design and realisation of an Italian historical building. Therefore, I asked the help of my sister Marzia, who is an art historian and lover, and I explained her the main ideas of this thesis, such as, for example, the use of patterns in the design and the importance of selecting a suitable language for the design purpose. Her suggestion is shown in the cover: the *façade of Santa Maria Novella* (1456-1470) in Florence, designed by *Leon Battista Alberti* (1404-1472).



Santa Maria Novella was built on the site of the previous *Santa Maria delle Vigne* (9th century). In 1221, this site was assigned to the Dominican Order that decided to build a new church, which was then called *Novella (New)*. The construction started around 1246 and when the church was finished, approximately in 1360, only the lower part of the façade was completed. In 1456,

Leon Battista Alberti designed the upper part of the façade on a commission from the local textile merchant Giovanni di Paolo Rucellai. The realisation of the façade was completed in 1470.

Leon Battista Alberti was a Renaissance humanist polymath. As architect and writer, he wrote the *De Re Aedificatoria*, a treatise on architecture based on the book *De Architectura* of the classical Roman writer *Vitruvius*. De Re Aedificatoria was the first architectural treatise of the Renaissance and, after its publication in 1485, became a major reference for architects. What a suitable reference also for a thesis about (model-driven, service-oriented, reference) architectures! Alberti regarded mathematics as the common ground of art and science, and emphasised the role of symmetry, proportion, geometry and the regularity of parts, like in the architecture of classical antiquity and, particularly, ancient Roman architecture. This recalled to me the idea of this thesis of using patterns as building blocks to facilitate the design and development process. I found this idea perfectly represented in the design of Santa Maria Novella shown in the (front) cover of this thesis.

One of the famous quotes of Alberti, together with the one in the back of the cover, is that "*a man can do all things if he will*". This quote summarises the concept of *universal man* promoted by the Reinassance, which considers humans empowered and limitless in their capability of development. In other words, humans should embrace all kind of knowledge and develop their abilities as fully as possible. I like this human-centric vision and I find it relevant to this thesis, in which I refer to demanding users that are aware of the opportunities offered by the continuously evolving technologies, and to service providers that consequently have to offer a wide range of various, enriched and personalised services to their users.

As final remark, this thesis is also about languages and I found very interesting that Alberti was also a linguist, who wrote in Latin, but also promoted the adoption of the *vulgar* Italian, from which the *modern* Italian originated.