# ARCHITECTURAL SUPPORT FOR CONTEXT-AWARE APPLICATIONS: FROM CONTEXT MODELS TO SERVICES PLATFORMS

Telematica Instituut Fundamental Research Series

001 G. Henri ter Hofte, *Working Apart Together: Foundations for Component Groupware*
002 P. J.H. Hinssen, *What Difference Does It Make? The Use of Groupware in Small Groups*
003 D.D. Velthausz, *Cost Effective Network Based Multimedia Information Retrievel*
004 L. van de Wijngaert, *Matching Media: Information Need and New Media Choice*
005 R.H.J. Demkes, *COMET: A Comprehensive Methodology for Supporting Telematics Investment Decisions*
006 O. Tettero, *Intrinsic Information Security: Embedding Information Security in the Design Process of Telematics Systems*
007 M. Hettinga, *Understanding Evolutionary Use of Groupware*
008 A. van Halteren, *Towards an Adaptable QoS Aware Middleware for Distributed Objects*
009 M. Wegdam, *Dynamic Reconfiguration and Load Distribution in Component Middleware*
010 I. Mulder, *Understanding Designers, Designing for Understanding*
011 R. Slagter, *Dynamic Groupware Services – Modular Design of Tailorable Groupware*
012 N.K. Diakov, *Monitoring Distributed Object and Component Communication*
013 C.N. Chong, *Experiments in Rights Control: Expression and Enforcment*
014 C. Hesselman, *Distribution of Multimedia Streams to Mobile Internet Users*
015 G. Guizzardi, *Ontological Foundations for Structural Conceptual Models*
016 M. van Setten, *Supporting People in Finding Information: Hybrid Recommender Systems and Goal-Based Structuring*
017 R. Dijkman, *Consistency in Multi-viewpoint Architectural Design*
018 J.P.A. Almeida, *Model-Driven Design of Distributed Applications*
019 M.C.M. Biemans, *Cognition in Context: The effect of information and communication support on task performance of distributed professionals*
020 E. Fielt, *Designing for Acceptance: Exchange Design for Electronic Intermediaries*

# Architectural Support for Context-Aware Applications: From Context Models to Services Platforms

*Patrícia Dockhorn Costa*

Enschede, The Netherlands, 2007

CTIT Ph.D.-Thesis Series, No. 07-108
Telematica Instituut Fundamental Research Series, No. 021 (TI/FRS/021)

*Cover Design*: Studio Oude Vrielink, Losser and Jos Hendrix, Groningen
*Cover Photo:* *Photograph by Liam Delahunty*
http://www.liamdelahunty.com
Beatriz Milhazes, *Peace and Love*, 2006
Gloucester Road Tube Station, District and Circle Line
Underground London
*Book Design*: Lidwien van de Wijngaert and Henri ter Hofte
*Printing*: Universal Press, Veenendaal, The Netherlands

*Graduation commitee:*
*Chairman, secretary*: prof.dr.ir. A. J. Mouthaan (University of Twente)
*Promotor*: prof.dr.ir. C. A. Vissers (University of Twente)
*Assistant Promotors*: dr.ir. M. J. van Sinderen (University of Twente)
dr. L. Ferreira Pires (University of Twente)
*Members*: prof.dr. C. Atkinson (University of Mannheim)
prof.dr. T. Plagemann (University of Oslo)
prof.dr.ir. S. Heemstra de Groot (Technical University of Delft)
prof.dr.ir. M. Akşit (University of Twente)
prof.dr.ir. R. Wieringa (University of Twente)

CTIT Ph.D.-Thesis Series, No. 07-108
ISSN 1381-3617; No. 07-108
Centre for Telematics and Information Technology, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands

Telematica Instituut Fundamental Research Series, No. 021
ISSN 1388-1795; No. 021
ISBN 978-90-75176-45-2
Telematica Instituut, P.O. Box 589, 7500AN Enschede, The Netherlands

Copyright © 2007, Patrícia Dockhorn Costa, The Netherlands

*All rights reserved. Subject to exceptions provided for by law, no part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the copyright owner. No part of this publication may be adapted in whole or in part without the prior written permission of the author.*

# ARCHITECTURAL SUPPORT FOR CONTEXT-AWARE APPLICATIONS: FROM CONTEXT MODELS TO SERVICES PLATFORMS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. W.H.M. Zijm,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 19 december 2007 om 15.00 uur

door
Patrícia Dockhorn Costa
geboren op 11 februari 1978
te Belo Horizonte, Minas Gerais, Brazilië

Dit proefschrift is goedgekeurd door:
prof.dr.ir. C.A. Vissers (promotor), dr.ir. M.J. van Sinderen (assistent-promotor) en dr. L. Ferreira Pires (assistent-promotor).

# Abstract

Context-awareness has emerged as an important and desirable feature in ubiquitous applications. This feature deals with the ability of applications to utilize information about the user's environment (context) in order to tailor services to the user's current situation and needs.

This thesis aims at providing an integrated solution for the development of context-aware systems. The main objective is to facilitate the development of context-aware applications, focusing on two aspects: on offering *context modelling abstractions* and on providing infrastructural support by means of a *context handling platform*. The context modelling abstractions provide application developers with proper conceptual foundations that can be extended and specialized with specific application requirements. The context handling platform allows application functionality to be delegated to the platform, which reduces application development effort, time and, therefore, costs. This allows application developers to focus on their core business, instead of being bothered with application realization details.

As applications become more complex and interconnected, there is an increasing need for context modelling abstractions that are appropriate to: (i) characterize the application's universe of discourse; to (ii) support common understanding, problem-solving, and communication among the various stakeholders involved in application development; and to (iii) represent context unambiguously. In order to cope with context modelling requirements, we define a set of context modelling abstractions which are based on conceptual modelling theories, and are supported by developments in foundational ontologies.

Context-aware applications use and manipulate context information to detect high-level *situations*, which are used to adapt application behaviours. We propose a model-driven approach for the specification of situations in context-aware applications and introduce a rule-based approach to implement situation detection. In our approach, situations are specified using a combination of UML class diagrams and OCL constraints. We

support a wide range of situation types, which can be composed of more elementary kinds of context types. We discuss how to cope with distribution and to exploit it beneficially for context manipulation and situation detection. We employ a generic rule-based platform to support the derivation of situations in a distributed fashion.

Finally, we provide a mechanism that facilitates the dynamic configuration and execution of particular application behaviours with the context handling platform, at platform runtime. This mechanism is based on a mobile rule engine that autonomously gathers context and situation values from distributed context processing components. This engine accepts application behaviours written in *ECA-DL*, which is a domain-specific language developed by us for the purpose of effectively specifying context-aware reactive behaviours.

# Acknowledgements

I would like to express my gratitude to all those who have helped me throughout the PhD trajectory.

I start by thanking my promoter Chris Vissers for his impeccable supervision, helpful insights and great wisdom. I also would like to thank Luís Ferreira Pires and Marten van Sinderen who have been my daily supervisors since 2002, first for the master's, then for the PhD. In these past years, Luís and Marten have taught me so much! They have been a constant inspiration to me for their discipline, perseverance and dedication to science and education. I thank them for becoming the researcher I am today. I hope we continue working together for a long time, we are a great team!

I'm honoured to have prof.dr. C. Atkinson, prof.dr. T. Plagemann, prof.dr.ir. S. Heemstra de Groot, prof.dr.ir. M. Akşit and prof.dr.ir. R. Wieringa as members of my PhD committee. I thank each one of them for accepting the invitation.

I have collaborated with Giancarlo Guizzardi and Ricardo Neisse in two chapters of this thesis, namely the Context Modelling and the Case Study chapters, respectively. Giancarlo and Ricardo are impressive researchers, and I'm glad we have worked together. This thesis would not be the same without their collaboration. I thank them for that! In addition, during the PhD I have worked with Laura Daniele and Nieko Maatjes whose master's theses have contributed to my work. I'm grateful to have had them as students! I'm also grateful to my AWARENESS colleagues with which I have cooperated throughout those years. Finally, I would like to thank Rodrigo Mantovaneli for his amazing photoshop work with the cover of this book.

Living abroad far from family and in a different (sometimes clashing) culture can be difficult. I'm thankful to all my friends for helping me to overcome those difficulties and for making my life in the Netherlands so joyful. In this respect, I would like to especially mention some of the *old*

and *new* Macandrian friends (Diana, Maarten, Arjan, Maartje, Ronald, Aleks, Sander, Azita, Femke, Ivo, Laura, Dimitri, Danah, Kasia, Liga, Sha, Wahe, Sarah and so many others), the extended Brazilian family (Ricardo, Tiago, Rodrigo, Giancarlo, Renata, José Gonçalves, Luiz Olavo, Luciana, Pablo, Flávia, Sharon, Lia, Jaqueline, Ismênia), the members of the ASNA group (Dick, Maarten Wegdam, Tom, Laura, Annelies, Marloes, Cristian Tzolov, Aart, Eduardo, Remco Dijkman, Remco van de Meent, Pravin, Kamran, Hailiang, Teduh), and so many other interesting people I met along the way (Agung, Susan, Diego, Sonia, Elfi, Andrew, Martine, Maria, Sorin, Olga, Maarten Steen, Stanislav, Vania, Riemer, Valerie, Peter, etc.).

I would like to thank my colleagues at the Computer Science Department of the Federal University of Espírito Santo for making me feel part of the team so quickly. I'm looking forward to working with all of them!

None of this would be possible without the support of my dear family back in Muqui, my hometown. Against all the odds, my parents Carlos Alberto and Maria do Carmo managed to provide me, my sister Fernanda and my brother Pedro Henrique with the best education one can get. They have my profound love and great respect for all they have achieved. I thank my family for everything and for always being there for me!

Finally, I would like to thank my husband João Paulo for the most amazing years *under the low sky*! I thank him for the endless technical discussions at home, for his unconditional support and for always believing in me! *To him I dedicate not only this book, but also my heart*.

*Patrícia Dockhorn Costa,*
*Vitória, October 2007.*

# Contents


| | | |
|---|---|---|
| CHAPTER 1. | **Introduction** | **1** |
| | 1.1 Background | 1 |
| | 1.2 Motivation | 3 |
| | 1.3 Related Work | 5 |
| | 1.4 Objectives | 6 |
| | 1.5 Approach | 7 |
| | 1.6 Scope and Non-Objectives | 10 |
| | 1.7 Structure | 10 |
| CHAPTER 2. | **General Concepts** | **13** |
| | 2.1 Context | 13 |
| | 2.2 Context-Aware Applications | 16 |
| | 2.3 Service-Oriented Architectures | 21 |
| | 2.4 Context-Aware Service-Oriented Platforms | 27 |
| CHAPTER 3. | **State-of-the-Art** | **37** |
| | 3.1 Context Modelling | 37 |
| | 3.2 Middleware and Platforms | 46 |
| | 3.3 Applications | 53 |
| CHAPTER 4. | **Architectural Patterns and the Context Handling Platform** | **55** |
| | 4.1 Context-Aware Patterns | 55 |
| | 4.2 Event-Control-Action (ECA) Pattern | 56 |
| | 4.3 Context Sources and Managers Hierarchy Pattern | 61 |
| | 4.4 Actions Pattern | 66 |
| | 4.5 The Context Handling Platform | 69 |
| | 4.6 Platform Services | 74 |
| | 4.7 Platform Stakeholders | 78 |
| | 4.8 Discussion | 82 |

**CHAPTER 5. Context Modelling** **83**
5.1 Characteristics of Context 84
5.2 Foundational Ontologies 84
5.3 Foundational Context Concepts 89
5.4 Context Models 96
5.5 Context Situation Models 105
5.6 Context Information Models 111
5.7 Discussion 119

**CHAPTER 6. Situation Realization** **121**
6.1 From Situation Specification to Situation Realization 121
6.2 Rule-Based Systems 124
6.3 Rule-Based Situation Realization 129
6.4 Mappings 138
6.5 Distribution Issues 143
6.6 Discussion 152

**CHAPTER 7. Controlling Services** **155**
7.1 The Controller 155
7.2 ECA-DL 161
7.3 Realization of ECA-DL Rules in Jess 181
7.4 ECA rules and the Situation Detection Framework 198
7.5 Discussion 204

**CHAPTER 8. Case Study** **207**
8.1 The Healthcare Application 207
8.2 The Context-Aware Policy Management Application 225
8.3 The Healthcare Application Prototype 232
8.4 Discussion 240

**CHAPTER 9. Conclusions** **243**
9.1 General Considerations 243
9.2 Research Contributions 244
9.3 Future Work 249

**APPENDIX A Mappings from OCL to Jess** **253**

**APPENDIX B ECA-DL Lifetime Constraints** **263**

**APPENDIX C Mappings from ECA-DL to Jess** **265**

**APPENDIX D Case Study Jess Rules** **271**

**References** 277

**Resumo** 287

**Publications by the Author** 289

Chapter 1

# Introduction

This thesis proposes an integrated solution for the development of context-aware applications. The main objective is to *facilitate* the development of context-aware applications by means of *context modelling abstractions* and a *context handling platform*. The context modelling abstractions provide application developers with proper conceptual foundations that can be extended and specialized with specific application requirements. The context handling platform allows application functionality to be delegated to the platform, which reduces application development effort, time and, therefore, costs. This chapter discusses background information, presents the motivation of this thesis, outlines the main research objectives, and presents the approach adopted.

This chapter is organized as follows: section 1.1 discusses background information; section 1.2 motivates the work proposed in this thesis; section 1.3 discusses related work; section 1.4 outlines the main research objectives; section 1.5 presents the approach adopted in the research; section 1.6 elaborates on the scope of this thesis, and finally section 1.7 presents the structure of the thesis.

## 1.1 Background

Over the last years, we have seen an unprecedented adoption of the Internet, communication and computing technologies everywhere in society. In parallel, advances in mobile computing, ubiquitous devices, software engineering, wireless and sensor technologies, offer the opportunity to introduce more sophisticated and user-friendly services [127]. Networked computing devices, getting increasingly smaller and combined with various sensing technologies, are being used to make physical environments "intelligent". Enabled by these developments,

computation is increasingly embedded in our social activities and physical environments [81].

Several fields of research have been combined to realize these advances, such as ubiquitous computing (often called pervasive computing), in combination with mobile and distributed computing, sensor networks, wearable computing and human-computer interaction. The following paragraphs briefly introduce some of these areas of research [21, 126]:

- Ubiquitous computing, or pervasive computing, supports the vision in which computing is transparently integrated into our living environment and daily lives. Everyday objects are empowered with computational capabilities in order to enable users to interact with computing devices more naturally and casually than we currently do with desktop computers [124, 126]. Several research topics collaborate to realize the ubiquitous computing vision, including mobile computing, distributed computing, sensor networks, service-oriented computing and artificial intelligence;
- Distributed computing is concerned with the coordinated use of physically distributed computers and applications. It considers issues such as communication transparency, data consistency and concurrency, system scalability, asynchrony and heterogeneity;
- Mobile computing refers to the capability of embedding computational power in portable and mobile computers and communication devices. It relies on the use of wireless technologies, such as WiFi, GSM and GPRS to enable wireless communication among devices;
- Sensor networks refer to networks of many, physically distributed sensor devices which monitor environmental conditions, such as temperature, humidity and pollution. This research field is concerned with how these devices should cope with severe limitations on battery power, computational capabilities, bandwidth and memory;
- Wearable computing refers to a field of research in which small, portable computers are designed to be worn on the body during use. This topic encompasses various lines of research such as user interface design, pattern recognition, use of wearables by disabled people, electronic textiles and fashion design.

Until recently, computation was limited to an interaction style in which users provide to a desktop computer all the required input to perform particular tasks. Nowadays, computation is evolving to an interaction style in which explicit user intervention is gradually less required. Explicit user inputs are being progressively replaced by conditions detected by sensors, devices and computers distributed over the environment.

Technological advances in the areas mentioned above support (or enable) the shift of computing from the desktop paradigm into a paradigm in which computing is immersed in the dynamic world where we live and

act [81]. This paradigm shift poses many challenges, mainly related to *whom*, *how*, *when* and *where* to deliver services in the myriad of situations that can be encountered in the real world, which is the ever-changing *context* of use. In this view, capturing and monitoring the context in which services should be delivered, and adapting services according to the context and users' preferences are essential requirements.

In this thesis we are particularly interested in applications that are capable of autonomously adapting their behaviour in response to context changes. These applications are called *context-aware applications*. Currently, many applications are already capable of using location as a context parameter for service adaptation, but an increasing number of opportunities are appearing to enrich the users' context, such as the users' current activities and medical conditions.

We define context as *the set of, possibly interrelated, conditions in which an entity exists* [79]. This definition reveals that context is only meaningful with respect to a thing that "exists" called *an entity*. The process of identifying relevant context consists of determining the circumstances ("related conditions") of an entity (or entities) that are relevant for the service provisioning.

## 1.2 Motivation

The following tele-monitoring scenario [6] illustrates the potential benefits of context-awareness in the medical domain.

*"Mr. Janssen is an epileptic patient and despite his medications, he still suffers from seizures. Because of his medical condition, Mr. Janssen is unemployed, home-bound, and his situation requires constant vigilance to make sure healthcare professionals are alerted of a severe seizure. Recently, Mr. Janssen has been provided with a tele-monitoring context-aware application capable of monitoring epileptic patients and providing medical assistance moments before and during an epileptic seizure. Measuring heart rate variability and physical activity, this application predicts seizures and contacts nearby relatives or healthcare professionals automatically. In addition, Mr. Janssen can be informed moments in advance about the seizure, being able to stop ongoing activities, such as driving a car or holding a knife. The aim is to provide Mr. Janssen with both higher levels of safety and independence allowing him to function more normally in society despite his disorder."*

Context-aware applications differ from traditional applications since they use sensed information to adapt the service provisioning to the current context of the user. In order to achieve that, context-aware applications, in general, should be capable of:

- *Sensing context from the environment*. The tele-monitoring application, for example, should be capable of sensing Mr. Janssen's physical conditions, such as heart rate, as well as sensing his current location and the location of relatives and healthcare professionals;
- *Observing, collecting and composing context information from various sensors*. In the tele-monitoring scenario, for example, the application should be capable of observing Mr. Janssen's heart rate variations and inferring whether he is having an epileptic seizure. Furthermore, the system may collect location information from various sensors to detect the closest healthcare professionals and relatives;
- *Autonomously detecting relevant changes in the context*. The tele-monitoring application should autonomously detect the occurence of an epileptic alarm;
- *Reacting to these changes, by either adapting their behaviour or by invoking (composition of) services*. Upon a seizure alarm, the tele-monitoring application should invoke services to contact the closest available healthcare professionals and relatives; and
- *Interoperating with third-party service providers*. Location and epileptic seizure alarms may be context information offered by services designed and implemented by third-party stakeholders. In addition, services used to contact Mr. Janssen's relatives and healthcare professionals are most probably offered by third-party telecommunication providers.

These and other characteristics pose many challenges to the development of context-aware applications, such as:

1. Support for context modelling abstractions that are appropriate to promote common understanding, problem-solving, and communication among the various stakeholders involved in application development [51];
2. Bridging the gap between information sensed from the environment and information that is actually syntactically and semantically meaningful to the applications;
3. Gathering and processing context information from distributed context services;
4. Dynamically adapting application behaviour (reactively and proactively); and,
5. Tailoring service delivery as needed by the user and his context.

These challenges require proper abstractions and methodologies that facilitate the development process of context-aware applications. In the particular case of large-scale context-aware applications, the following aspects have so far been (partially) neglected in the literature:

- As applications become more complex and interconnected, there is an increasing need for abstract context modelling techniques to facilitate the specification of context models that are clearer and easier to

understand. What are the fundamental concepts that can be beneficially used in context modelling?

- Information from several physically distributed context services may be aggregated and interpreted to yield newly produced context information, at application runtime. How to address runtime context information aggregation and interpretation?
- In a ubiquitous computing world where the environment is equipped with all kinds of sensors, applications may profit from context services that are unknown to the application beforehand. How to address ad hoc networking of context services?

## 1.3 Related Work

Much effort has been spent to tackle the challenges mentioned earlier. The work presented in [56, 58], for example, introduces a conceptual framework and an infrastructure for context-aware computing, partially addressing challenges 1, 2 and 5. The basis of this work is a formal, graphical context-modelling technique called CML (the Context Modelling Language). CML extends Object-Role Modelling (ORM), which uses *fact* as the basic modelling concept, and the *situation* abstraction to leverage on facts to represent high level context information.

The framework presented in [7] proposes a rule-based sentient object model to facilitate context-aware development in an ad-hoc environment. The main functionality is offered in a tool that facilitates the development process by offering graphical means to specify context aggregation services and rules, tackling, to some extent, challenges 3 and 4.

The work presented by [16] proposes a *broker* architecture for pervasive context-aware systems (CoBrA). This architecture aims at addressing context representation, sharing, reasoning and privacy aspects, by means of an intelligent agent called context broker. The work is based on semantic web technologies, such as OWL and RDF, and it addresses, to some extent, challenges 1, 2 and 3.

The context toolkit presented in [24] provides a set of abstractions that can be used to implement reusable components for context sensing and interpretation, partially tackling challenge 2. This work has pioneered in proposing generic support for context-aware application development by means of a conceptual framework. Since it has been presented in the early stages of context-aware computing, several issues later identified, such as the need for proper context modeling abstraction and adaptation support, were not considered in [24].

Although much work has been proposed to address some of the challenges we mentioned, an integrated solution aiming at tackling all of

them has not yet been provided. In addition, most of the approaches in the literature are technology specific, focusing on solutions to specific problems. Few approaches consider generic solutions, which aim at providing conceptualizations and reusable architectural patterns.

## 1.4 Objectives

In this thesis we aim at providing an integrated solution for the development of context-aware systems by addressing the main challenges mentioned earlier.

The dynamic nature of context-aware applications, and the increasing integration of these applications into our daily tasks in a variety of domains (e.g., home, work and leisure), generate rapid changes in the requirements for the technology to support these applications. Although it is not possible to fully predict these changes, the supporting technology can be designed in such a manner that it can be configured to match changing requirements, preferably at system runtime. This calls for a high level of flexibility, which increases development costs and complexity. It is not cost-effective to build each individual application from scratch. For reasons outlined in the previous section, it is too complex for each individual application to capture and process context information just for its own use.

We aim at coping with these issues by means of a shared *Context Handling Platform* to support context-aware applications. This platform comprises, among others, reusable context processing and management components, which facilitate context-aware application development and deployment. It provides building blocks that can be combined and specialized to satisfy specific application requirements. For example, the platform provides special components that can take application-specified rules and procedures as input in order to carry out application-specific adaptation within the platform. Therefore, flexibility and adaptability are characteristics of strategic importance in the Context Handling Platform.

In order to carry out application-specific adaptation, the platform needs to properly understand the application's universe of discourse. In particular, the platform should be able to understand and reason about the context situations that are of particular interest to these applications. In order to cope with this issue, we propose basic conceptual foundations for context modelling, which allow designers of context-aware applications to represent relevant elements of a context-aware application's universe of discourse. Further, we present a situation-based approach that allows the specification of context situations.

Furthermore, the platform may incorporate reusable services, which are externally developed and provided by various third-party stakeholders. The

context models produced using our conceptual foundations are also beneficially applied to promote common understanding and communication between the platform and these third-party services.

The objective of this thesis is to facilitate the development of context-aware applications. We mainly focus on two aspects: offering context modelling abstractions and providing infrastructural support by means of a Context Handling Platform. Our context modelling abstractions provide application developers with proper conceptual foundations that can be extended and specialized with specific application requirements. The context handling platform allows application functionality to be delegated to the platform, which reduces application development effort, time and, therefore, costs. This allows application developers to focus on their core business, instead of being bothered with application realization details.

These objectives have been split into three sub-objectives, targeted to particular stakeholders:

- To provide context-aware platform developers with a reference architecture that can be extended and specialized to specific target architectures;
- To provide context-aware application developers with a Context Handling Platform in which context-aware applications can be developed and deployed upon. This platform is an instance of the reference architecture being proposed; and
- To provide context-aware application developers with context modelling abstractions that serve as a foundation to be extended and specialized with application specific requirements. These modelling abstractions are appropriate to (i) define the application's universe of discourse; (ii) specify particular situations, which are state-of-affairs of interest to applications; and (iii) specify context events, which are characterized by situation transitions.

## 1.5 Approach

In order to achieve our goals, we begin by performing a state-of-the-art survey on context-aware systems in general, identifying the main characteristics, requirements and the concepts involved in context-aware application development. Based on this survey, we identify the requirements for modelling context and for building a generic and flexible platform to support context-aware application development. The following topics are relevant for the approach taken in this thesis.

### 1.5.1 Context modelling

In order to cope with context modelling requirements, we define a set of context modelling abstractions that is based on conceptual modelling theories [80]. As a basic distinction, we propose the separation of the concepts of *entity* and *context*. We also propose that context should be characterized as either *intrinsic* or *relational*. Further, we define a *situation-based* approach which allows application designers to compose primitive elements of the entities' contexts, to yield more complex situations. We motivate our concepts by relating them to developments in foundational ontologies [51], which are in line with conceptual theories in the areas of philosophy and cognitive sciences.

Following the application design process, we provide support for bridging the gap between conceptual context models and *context information models*. In the scope of context information models, we should refer to *context information* as opposed to context. Context information refers to the representation of (constituents of) context in an application, such that this representation can be manipulated and exchanged. Situations in context information models are detected based on context information. Issues that become relevant for context information models relate to: (i) how context is sensed; (ii) how context information is produced, learned, inferred and used, and (iii) the validity and quality of context information.

In order to demonstrate the *feasibility* of the context modelling approach, we propose a possible realization alternative based on a model-driven approach, which enables context and situation realization elements to be derived systematically from specification elements. We introduce a novel rule-based situation realization alternative, which executes on mature and efficient rule engine technology available off-the-shelf.

### 1.5.2 Context handling platform

Inspired by our state-of-the-art survey, we outline recurring problems that appear in context-aware application development. For each of these recurring problems, we propose architectural patterns that contain a solution scheme, focusing on structural aspects with components, their relationships and dynamic (behavioural) aspects. Using these architectural patterns, we identify components that can be generalized and included as part of the platform. These components form the platform building blocks that are composed to meet application-specific requirements. For example, using the *Event-Control-Action* (ECA) architectural pattern (Chapter 4) we identify the need to distinguish the tasks of gathering and processing context information from tasks of triggering actions in response to context changes, under the control of an application behaviour description.

Once the generic components are identified, we are capable of defining a complete *context handling platform*. This process includes the identification and description of stakeholders involved in the platform development and lifetime (Chapter 4).

### 1.5.3 Controlling services

Our context handling platform should be capable of providing (i) *flexibility* to allow application-specific behaviours to be deployed at platform runtime; (ii) *extensibility* to allow newly defined services and application-specific reactive behaviours to be incorporated in the platform on demand, at runtime; and (iii) *adaptability* to allow application-specific adaptation according to changes in the context to be handled by the platform. We aim at tackling flexibility, extensibility and adaptability requirements by means of the *Controlling services*, which are offered by *controller components*.

Using our context and situation modelling theory, application developers are capable of defining the configuration information necessary to enable the platform to carry out application-specific functionality. This configuration information follows a rule-based approach, which is in line with the Event-Control-Action (ECA) pattern mentioned earlier. These rules are called *Event-Condition-Action (ECA)* rules, and are composed of three parts:

- An event part, which allows complex compositions of events that represent any occurrences of interest in the user's context;
- A condition part, which allows various combinations of context and situation conditions;
- An action part, which allows the specification of service invocations.

We develop a domain-specific language, coined ECA-DL, to allow the specification of ECA rules. This language incorporates temporal aspects, empowering application developers to define complex relationships between situations, such as their temporal ordering and causal relations. Since this language is specially developed for the purpose of specifying context-aware application behaviours, it is expected to be more appealing and easier to use than other general purpose languages.

Application-specific ECA rules are provided as input to controlling services, at platform runtime. In order to process and execute ECA rules, the controller component uses discovery services, and performs composition of *context provisioning* and *action services*. Context provisioning services aim at providing information to reason about the context and situations. Action services implement the actions to be triggered when certain events occur and the context conditions are satisfied. In order to detect events, the controller component (i) observes context and situation information from context provisioning services; (ii) determines, based on

the observed context and situation information, whether situations hold or not; and (iii) reasons about situation transitions.

### 1.5.4 Case study

Finally, as proof of concept, we demonstrate the feasibility of our approach by developing context-aware applications using our context modelling abstractions in combination with our proposed platform support. For each of the case studies, we outline the design process following the development approach proposed. In order to demonstrate the feasibility in a realistic application, we have built an application prototype that implements the design products obtained from the design activities. In addition, this prototype allows us to critically assess our approach with respect to performance and scalability issues.

## 1.6 Scope and Non-Objectives

In this thesis we focus on context-aware mobile computing systems and context-aware networked applications that are widely distributed. We consider the notion of context applied to the users of services provided by context-aware applications and platforms, as opposed to computational context or network resources context.

Our intention is to provide general support for developing context-aware applications focusing on (i) conceptual context modelling, (ii) application reactive behaviour modelling, and (iii) infrastructural support. Furthermore, as part of the context handling platform development, we address distribution, performance and scalability aspects by means of situation reasoning distribution, as illustrated in Chapters 5, 6 , 7 and 8.

In this thesis we do not extensively address (i) privacy, security and trust issues, (ii) quality of context information, (iii) usability aspects of context-aware applications, (iv) sensor technology, and (v) specialized reasoning algorithms and computational models.

## 1.7 Structure

The structure of this thesis reflects the research approach we adopted to achieve our goals. The remainder of this thesis is structured as follows:

- *Chapter 2 – General Concepts*. This chapter presents the basic terminology and the fundamental concepts used throughout this thesis. Issues like the distinction between *context* and *context information* are addressed. We

further introduce the notion of *service-oriented architecture*, which describes the design principle underlying our *context handling platform*;

- *Chapter 3 – State-of-the-Art*. This chapter presents the state-of-the-art in context-awareness. Our discussion focuses on three main branches of research, namely *context modelling*, *middleware and platforms*, and *applications*;
- *Chapter 4 – Context-Aware Architectural Patterns and the Context Handling Infrastructure*. This chapter presents the architectural patterns we have proposed to support the development of context-aware platforms and applications. These patterns present solutions for recurring problems associated with managing context information and proactively reacting upon context changes. We demonstrate the benefits of applying these patterns by discussing our *context handling platform architecture*. In addition, this chapter elaborates on the stakeholders involved in the platform development and commercialization;
- *Chapter 5 – Context Modelling*. This chapter presents the modelling abstractions we propose to model context and situations. We focus on providing modelling foundations that can be extended with application-specific aspects;
- *Chapter 6 – Situation Realization*. This chapter aims at demonstrating the *feasibility* of our situation modelling approach by proposing a rule-based realization alternative. This solution is based on the use of a general-purpose rule-based platform, which guarantees the efficiency of situation detection;
- *Chapter 7 – Controlling Services*. This chapter addresses flexibility, extensibility and adaptability aspects of the context handling platform. We present a rule-based approach that can be used to configure the platform in a flexible manner. We also define ECA-DL, which is a rule-based language used for the specific purpose of defining reactive behaviours of context-aware applications;
- *Chapter 8 – Case Study*. This chapter demonstrates the *feasibility* of the development approach proposed throughout chapters 2 to 7 by means of two design examples. The application scenarios considered are the *healthcare*, and the *policy management* scenarios. These scenarios deal with different application requirements, which demonstrate the suitability of the context handling platform to support applications in different domains. We choose one of the design products to be prototyped, namely for the healthcare scenario. With the prototype we collect measurements in order to assess scalability and performance issues on a realistic context-aware application;
- *Chapter 9 – Conclusions*. In this chapter, we make a critical reflection on the work presented in this thesis. We outline our most important

achievements and discuss the encountered drawbacks. Finally, we identify the topics that require further investigation.

*Figure 1-1* schematically depicts the structure of this thesis.

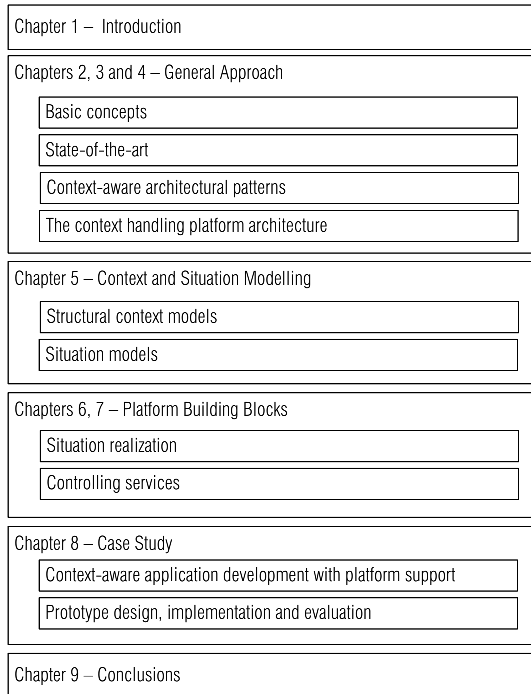



*Figure 1-1* Structure of the thesis

Chapter 2

# General Concepts

This chapter presents the basic terminology and the fundamental concepts used throughout this thesis. Issues like the distinction between *context* and *context information* are addressed. We further introduce the notion of *service-oriented architecture*, which describes the design principle underlying our *Context Handling Platform*.

This chapter is organized as follows: section 2.1 introduces basic concepts and the terminology related to *context*; section 2.2 elaborates on the concept of context-aware applications; section 2.3 discusses service-oriented architectures, and finally section 2.4 introduces our context handling platform.

## 2.1 Context

Context is a concept that appears in various disciplines [85]. In philosophy and cognitive sciences, Davies and Thomson [27] highlighted the importance of understanding and studying context since "organisms, objects and events are integral parts of the environment and cannot be understood in isolation of that environment". In linguistics [43, 95], researchers seek to understand the impact of context in, for example, choosing utterance style and utterance interpretation. In psychology [112, 128], researchers are interested in how changes in context affect various cognitive processes, such as perception, reasoning, decision making, and learning.

In computer science the notion of *context* has been first mentioned in artificial intelligence [49, 73]. In this field, context appears as a means of partitioning a knowledge base into manageable sets or logical constructs that facilitate reasoning activities. More recently, with advances in mobile computing, context has become a topic of interest to other areas of computer science, such as telematics and ubiquitous computing. In these

areas, context is usually regarded as the set of environmental conditions that can be used to adapt mobile applications to their user's current situation and needs. These areas are particularly interested in context due to mobility, which generates opportunities to explore context: since the context of the user changes frequently, applications running on the user's mobile devices may adapt their behaviour according to these changes.

In this scope, definitions of context found in the literature focus on either the application or the application's user, as indicated by [85]. With the term application's user we refer to the "end-user", i.e. a human being that uses the application. Chen and Kotz [18], for example, define context from the perspective of the application. They regard context as the "set of environmental states and settings that either determines an application's behaviour or in which an application event occurs that is interesting to the user". Conversely, Dey, Abowd, and Wood [25] define context as "the user's physical, social, emotional, or informational state", thus focusing on the application's user, as opposed to focusing on the application. Similarly, Zetie [129] describes context as "the knowledge about the goals, tasks, intentions, history and preferences of the user that a software application applies to optimize the effectiveness of the application". The most referred definition of context given by Dey et al. [22] balances the focus on both, users and applications: "context is any information that can be used to characterize the situation of entities (i.e. whether a person, place, or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves".

### 2.1.1 Context versus context information

The definitions given above actually refer to *context information* as opposed to *context*. We distinguish these concepts in our approach. We regard context as the real world phenomena, while context information refers to the representation of (constituents of) context in an application, such that this representation can be manipulated and exchanged. In section 2.2.2 we further elaborate on the concept of context information.

In this thesis we consider the following definition of context [79]:

***Context:*** *the set of, possibly interrelated, conditions in which an entity exists.*

This definition reveals that context is only meaningful with respect to a thing that "exists", which we call here an *entity*. The concept of entity is fundamentally different from the concept of context: context is what can be said about an entity in its environment, i.e. context does not exist by itself. Examples of entities are persons, computing devices and buildings. The context of an entity may have many constituents, which are defined here as

the “possibly interrelated conditions", or just *context conditions*. Context conditions reflect particular aspects of the circumstances in which entities exist. Examples of some context conditions of a person are the person’s location, mental state, and activity. Together, these context conditions form the entity’s context.

### 2.1.2 Context modelling

The process of identifying relevant context consists of determining the “conditions” of entities in the application’s universe of discourse (e.g., a user or its environment) that are relevant for a context-aware application or a family of such applications.

***Context model:*** *the representation of context conditions or circumstances which are relevant for a context-aware application or a family of such applications.*

***Context modelling:*** *the process of producing context models.*

We distinguish two modelling phases in our context modelling approach, namely, *conceptual context modelling* and *context information modelling*. Conceptual context modelling aims at producing models which are conceptual models of context. Conceptual models are, in the sense of [51, 80], representations of a “given subject domain independent of specific design or technological choices”. In this scope, only the concept of context (as opposed to context information) is relevant, since these models abstract from how context is sensed, provided, learned, produced and/or used. Context information modelling, on the contrary, aims at delivering models that extend the conceptual models of context by introducing technology aspects to the model, such as how context is sensed, gathered or learned.

Although only few definitions of context in the literature explicitly distinguish between context and context information, and therefore also between conceptual context modelling and context information modelling, we argue that this distinction is fundamental in the development of context-aware applications. We justify this argument based on the importance of conceptual modelling in the application design process, as emphasized by [51, 80]: “conceptual specifications are used to support understanding, problem-solving, and communications, among stakeholders about a given subject domain. Once a sufficient level of understanding and agreement about a domain is accomplished, then the conceptual specification is used as a blueprint for the subsequent phases of a system’s development process”. Therefore, the quality of context-aware applications depends on the quality of the conceptual context models upon which their development is based.

We believe that conceptual modelling of context should precede the detailed design of context-aware applications, in a similar way as analysis should precede the detailed design of an information system.

## 2.2 Context-Aware Applications

We discuss relevant aspects of context and context-aware applications in the sequel.

### 2.2.1 Definition

*Figure 2-1* depicts an intuitive view of a user in his/her context, and a context-aware application (focusing on a single user only). We define a context-aware application as follows.

***Context-aware application:*** *a distributed application whose behaviour is affected by its users' context.*

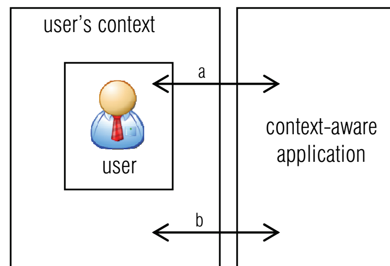



*Figure 2-1* Intuitive view of a context-aware application interacting with a user and his/her context

In *Figure 2-1*, the arrow "a" shows that the user and the context-aware application interact. Similarly, the arrow "b" shows that the user's context and the context-aware application interact.

The interactions represented by arrow "a" enable, for example, user's input to be provided to the application, such as user commands and preferences, and the use of the service delivered by the context-aware application. The interactions represented by arrow "b" enable the context-aware application to capture particular context conditions from the user's context.

As depicted in *Figure 2-2*, interactions take place at intersection points, which are shared mechanisms between interacting entities. Two types of shared mechanism are shown in this figure: one between a user and the context-aware application, and the other between the user's context and the context-aware application. These shared mechanisms are represented in *Figure 2-2* by the intersections between a user and the context-aware

application (ip-$a_1$.. ip-$a_n$), and between his/her context and the context-aware application (ip-$b_1$.. ip-$b_n$), respectively. Each of these intersections symbolizes an *interaction point (ip)* [40, 106].

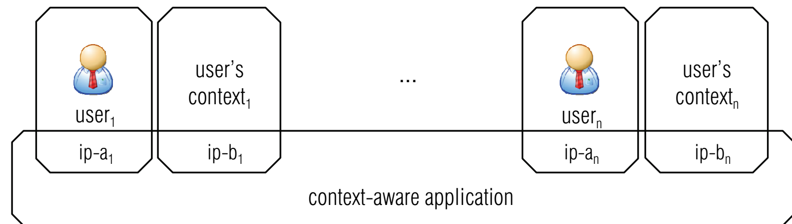



*Figure 2-2* A context-aware application, its user and her/his context, and the shared mechanisms for interaction

The intersection between a user's context and a context-aware application (ip-$b_1$.. ip-$b_n$) includes sensors that detect the context conditions used by applications to respond accordingly. An example of a sensor which is useful for context-aware applications is a Global Positioning System (GPS) device, which can be used to continuously track a user's location. Context information exchanged in interactions with the user's context consists of geographical coordinates for the user's current location. Another example of a sensor is a body thermometer, which can be used to monitor a patient's body temperature. Context information exchanged in interactions with the user's context in this case consist of the temperature measurements in degrees Celsius.

*Figure 2-3* depicts the graphical representation used in this thesis, which is defined in [40, 106]. An interaction point is expressed by oval shapes that overlap with the entities that share the interaction point. Interactions taking place in interaction points ip-$a_i$ and ip-$b_i$ model the activities performed in cooperation between the user and the context-aware application, and between the user's context and the context-aware application, respectively. As defined in [40, 106], an "interaction models the *establishment* of the result, abstracting from the possible complex mechanism that lead to the establishment of the result".

All possible information types referring to context that may be established in both ip-a and ip-b are defined in a *context information model*.

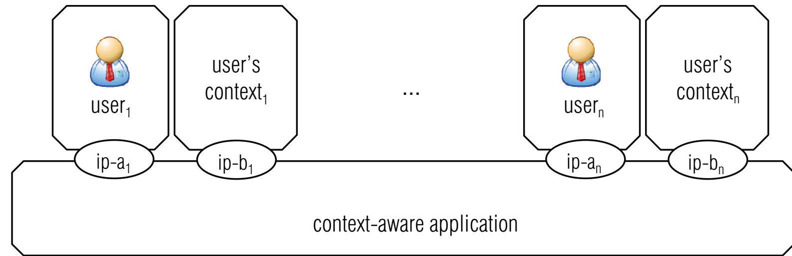



*Figure 2-3* Entities with shared interaction points

A user is always complemented by his/her context. Therefore, the user and his/her context are always paired. Although in the real world users interact with each other and their contexts influence each others' contexts, these (inter)relationships are not modelled by the context-aware application. Therefore, from the context-aware application point of view, information established in any paired interaction points (ip-$a^i$, ip-$b^i$) is independent from information established in (ip-$a^{i+}{}_1$, ip-$b^{i+}{}_1$), and so forth.

### 2.2.2 Capturing context

In our definition of context-aware application we do not distinguish between manually provided information and automatically acquired (sensed) information. We prefer to give a wider definition of context information, which encompasses a broader range of context information types, regardless whether information is sensed or manually provided. Our intention is to provide guidelines (by means of our context modelling approach) for identifying context information types, rather than fixing pre-defined sets of context information types, or prescribing whether information is acquired or not.

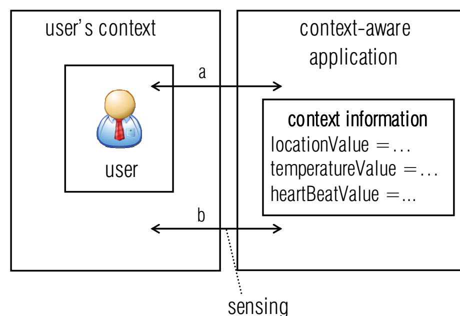
It is actually the responsibility of the application designer to decide whether context information is relevant to the application, since this decision depends on the application's universe of discourse and the application's state-of-affairs of interest. In addition, the computational capabilities (sensor technology) available for building the context-aware application typically determine the constraints for acquiring context automatically or manually. Whether manually provided or automatically acquired, we assume context information is always provided to applications through interaction points of type ip-b (see *Figure 2-3*).

In this sense, the definition of context information mentioned in section 2.1 given by Dey et al. [22] suffices, since it allows one to (i) abstract from whether information is manually or automatically acquired, and (ii) define a broad range of context information categories. In this thesis we rely on this definition of context information.

When considering context-aware applications, context consists of possibly interrelated conditions in the real world (represented in our figures

by the user's context). Applications still need to quantify and capture these conditions in terms of context information in order to reason about context. *Figure 2-4* shows an intuitive representation of the user's context conditions in the context-aware application by means of *context information*. Context information values are depicted inside the context-aware application. These values are approximations of the real world context conditions of the user's context. Only specific context conditions are relevant to the context-aware application. The process of identifying relevant conditions is part of the context modelling activities.

*Figure 2-4* Intuitive view of context in the real world (user's context) and context information in a context-aware application




### 2.2.3 Situations

Modelling the context conditions in the application's universe of discourse allows application designers to represent all possible state-of-affairs in the application's universe of discourse without discriminating particular situations that may be of interest to applications. For example, while we may capture in a context model that a person may be married to another person, it is not the objective of the context model to make statements about particular instances of persons. Therefore, we do not explicitly represent in a context model that John is married to Alice, or that John has been married to Alice for 10 years. In order to enable the representation of particular state-of-affairs, we introduce the concept of *situation*. Situations are defined as follows:

***Situation:*** *particular state-of-affairs that is of interest to applications. A situation is a composite concept whose constituents may be (a combination of) entities and their context conditions.*

Situations build upon context models since they can be composed of more elementary kinds of context conditions, and in addition can be composed of existing situations themselves. Examples of situations that may be of interest to a context-aware application are "user is running and he/she has access to

his/her mobile phone", and "user is in danger of an eminent epileptic seizure and he/she is driving". The first situation combines the context conditions "user is running" and "user has access to mobile phone". The second situation combines the context conditions "user having an epileptic seizure" and "user driving a car".

In our approach we define *situation types*, which aim at characterizing situations with similar properties. For example, the situation type "John is within 50 meters from Alice" consists of all situation instances in which the distance between John's and Alice's location is smaller than 50 meters. Similarly, the situation type "Person is within 50 meters from another person" consists of all situation instances in which the distance between any two persons' location is smaller than 50 meters.

A situation exhibits temporal properties, such as the time interval during which the situation holds. As an example, consider the situations "John is married to Alice" and "John and Alice are divorced". From time $t_0$ to $t_{10}$ (e.g., for 10 years) John has been married to Alice. During this interval, at any time, the situation "John is married to Alice" holds. In this way, we can also define temporal operations for relating situations according to their occurrence intervals, such as precedence, overlapping, and post-occurrence.

The process of defining situations and situation types is part of the context modelling activities, which are discussed in Chapter 5.

### 2.2.4 Quality of context

Context-aware applications depend strongly on the availability and the quality of sensors, or more generally, *context sources*. Context-aware applications also depend on the availability and capabilities of portable (mobile) devices that can be used to interact with the user. In the last years, sensors and devices of higher quality are proliferating due to advances in hardware and miniaturization. However, current sensors and devices are not sufficiently accurate, and they may introduce noise, delays and imperfections to the context information being sensed and/or measured.

Therefore, the quality of context information is strongly dependent of the mechanisms used to capture the corresponding context conditions from the user's environment. Some context conditions may have to be measured, and the measuring mechanisms may have a limited accuracy; other context conditions may vary strongly in time, so that the measurement may quickly become obsolete. Decisions based on context information taken in context-aware applications may also consider the quality of this information, and therefore context-aware applications also need meta-information about the context condition values, revealing their quality.

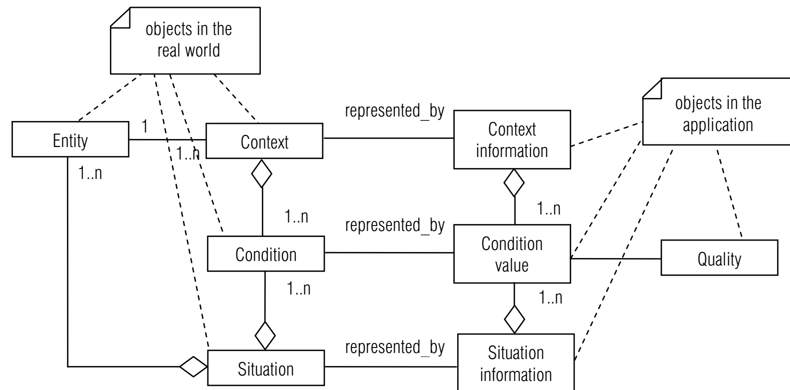



*Figure 2-5* Diagram of the concepts related to context (real world and applications)

*Figure 2-5* depicts a UML class diagram that summarizes the concepts presented so far. This model represents context in the real world (user's environment) and in applications (as context information). Context always refers to an entity, and can be represented by a collection of conditions. In context-aware applications, context is referred to as context information, and its corresponding conditions are represented by condition values in the applications. In addition, these context condition values are related to some quality measures, which determine the quality of the information (e.g., how precise, accurate or fresh the information is).

The figure also represents situations, which are particular compositions of entities and context conditions. Context conditions in a situation may belong to different entities. A situation is represented as situation information in context-aware applications.

Although we discuss context and situation information from the point of view of condition values, context modelling can only be re-used and generalized when (i) condition and situation types, and (ii) their semantics and relationships are clearly defined. Chapter 4 focuses on context modelling issues, which also include situation modelling.

## 2.3 Service-Oriented Architectures

Service-Oriented Architectures (SOAs) use an architectural style centred on the concept of service that can be applied in the design of distributed applications [77]. In this thesis, we adopt this architectural style for the design of context-aware applications. In this sense, we can say that a context-aware application provides a *context-aware service* to its users.

In order to understand the concept of service we consider the following definition of system [79]:

***System:*** *a regularly interacting or interdependent group of items forming a unified whole.*

This definition acknowledges that a system has an *internal perspective*, which consists of a group of items that are interdependent and interact regularly, but also an *external perspective*, which gives the system its identity as a whole. The *service concept* concentrates on the external perspective of a system, in terms of the behaviour that can be experienced by the environment (users) of a system. Therefore, a (complete) service specification consists of the interactions between the system providing the service and the users of this service, and the relationships between these interactions. This separates the service (the supported behaviour) from the entity providing the service (the system). *Figure 2-6* depicts a system as (a) a regular group of items and (b) a whole. *Figure 2-6* also shows the system's environment and the service this system provides. The internal perspective of the system is irrelevant for the definition of the service, as represented in *Figure 2-6* (b).

As opposed to the figures depicted in the previous sections, which represent the interacting entities, the figures presented in this section represent the behaviours exhibited by entities, and the relations between these behaviours. Behaviours are presented by oval shapes, while double-sided arrows represent interactions.

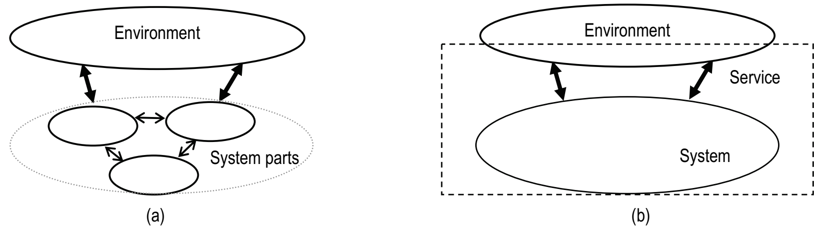



*Figure 2-6* A system's (a) internal perspective and (b) external perspective

In service-oriented architectures, applications are composed of application parts by considering these parts only from the point of view of the services that these parts provide. Application parts are often called *application components* (or simply *components*) in the literature. From now on we use the term *component* when referring to an application part. Components may not be necessarily implemented using typical component-based technologies, such as EJB [39] or CCM (CORBA Component Model) [87].

The service-oriented architecture approach implies that components make use of each other's services to cooperate in order to support the goals of the application. In addition, services are the only way to interact with components, enforcing in this way a discipline in the composition of the application. Furthermore, service interactions in a service-oriented

architecture are usually implemented using open standard technologies, as opposed to proprietary solutions.

*Figure 2-7* depicts an application as a composition of components that interact by invoking each other's services. Directional arrows represent relations of type *use*. The application controller component *uses* the services of the Context manager component and the Action performer component, and the Context manager component *uses* the services provided by the Context sources *i* components.

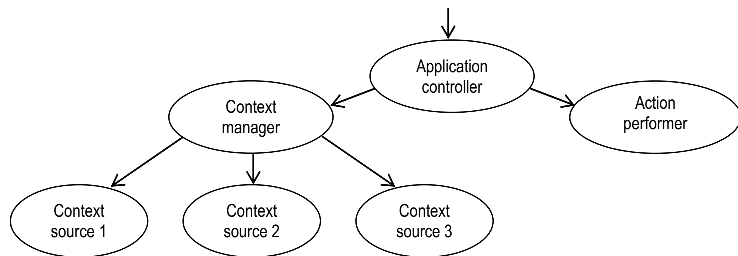



*Figure 2-7* Example of an application consisting of components that use each other's services

Service-oriented architectures are often seen as an evolution of component-based architectures. In component-based architectures, components are defined as the units of deployment of a distributed application, and can only be reached through their interfaces. Service-oriented architectures resemble this latter characteristic of component-based architectures, but do not necessarily preclude the use of component technologies. However, when component-based technologies are used in the implementation of service-oriented architectures, application components can be components in the sense of component-based architectures (units of deployment).

Although in the literature the term service-oriented architecture is strongly related to Web Services [125], there are many other technologies that can be used to implement service-oriented architecture. Examples of these technologies are Jini, Java RMI/EJB and CORBA. Service-oriented architecture is a design discipline of organising applications as composition of services. Service-oriented architectures are normally built on top of a middleware that supports the interactions between application components.

### 2.3.1 Service user and service provider roles

In most concrete service-oriented architectures there is a sharp distinction between the roles of service user and service provider. For example, in *Figure 2-7* the Context manager component plays the role of service user when interacting with the context sources *i*, but plays the role of a service provider when interacting with the Application controller component. The service of the Context manager component in this case is offered to the Application controller, but in order to support this service the Context

manager has to use the services supported by the Context sources *i* components.

The service concept can be applied recursively, in the sense that a system component can provide a service, but at the same time it can shield a whole composition of services from its service users. *Figure 2-7* also illustrates this situation, since the Context manager provides a service to the Application controller as can be seen from the external perspective by the latter application component. The designers of the Context manager know that in order to provide its service, the Context manager needs to interact with the Context sources *i*.

### 2.3.2 Service descriptions

Another important characteristic of service-oriented architectures is that services should be described using some suitable notation in order to be invoked by their users. Normally this description contains the definitions of the interactions that a service supports, and information on how to perform these interactions (e.g., the technology and address of the interface). Service descriptions can be published in directories, which also provide services themselves (registration and discovery services), allowing the potential service users to discover and access these services either at design-time or runtime.

*Figure 2-8* Typical roles in a service-oriented architecture

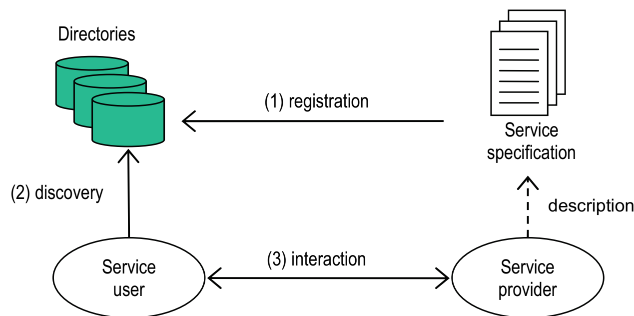



*Figure 2-8* depicts the sequence of messages between the service user, service provider and the directories. In order to publish the service, the service provider registers the service description with a service directory (message (1)). Potential users discover services of interest looking up in services directories (message (2)). Message (3) represents the actual service usage.

Often, middleware used in a service-oriented architecture offer basic services for service publication and discovery. For example, in Web Services architectures, UDDI repositories [120] are used as service directories, and in CORBA, the OMG trader [88] can be used for the same purpose.

### 2.3.3 Interaction patterns

Components in a service-oriented architecture may interact following different interaction patterns. Some common interaction patterns for service-oriented architectures include message passing, request-response, subscribe-notify and publish-subscribe.

The *message passing* interaction pattern [62] is realized by one-way message flow, sent by a service user, resulting in the conveyance of information from the service user to the service provider. *Figure 2-9* depicts a schematic view of the one-way interaction pattern.

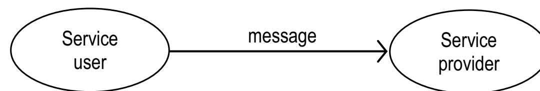



*Figure 2-9* One-message interaction pattern

The *request-response* interaction pattern [62] is realized by two related one-way message flows, one for the request message and one for the response. *Figure 2-10* depicts a schematic view of the request-response interaction pattern.

The request message, sent by a service user, results in the conveyance of information from that service user to a service provider, requesting a function to be performed by the service provider, followed by a second interaction, called response. The response is sent by the service provider, resulting in the conveyance of information from the service provider to the service user in response to the request. Request and response messages are always paired.

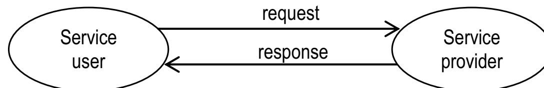



*Figure 2-10* Request-response interaction pattern

The *subscribe-notify* interaction pattern is realized by one-way request message and, eventually, multiple one-way response messages, called notifications. The request message is sent by the service user, requesting a function to be performed by the service provider. The response message may convey information to the service user in one of the following ways: time-based, or event-based.

Time-based notifications are sent by the service provider from time to time, in which the frequency of notifications should be specified in the request message, or be pre-defined in the service specification.

Event-based notifications are sent by the service provider every time an event of interest happens, i.e. the frequency of notification may vary according to the occurrence of events. The subscribe messages should identify the conditions under which the events are to be notified. This may require the use of, for example, a pre-defined subscription language.

Alternatively, conditions for event notifications may be fixed in the service specifications. *Figure 2-11* depicts a schematic view of the subscribe-notify mechanism.

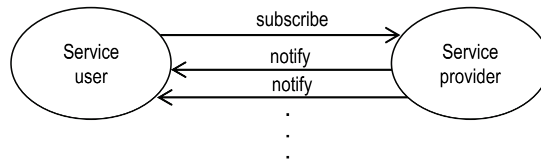



*Figure 2-11* Subscribe-notify interaction pattern

The *publish-subscribe* interaction pattern is a combination of the one-way message and the subscribe-notify interaction patterns. The publish-subscribe mechanism provides a loosely coupled form of interaction with respect to [38]:

- Space: interaction parties do not need to know each other location;
- Time: interaction parties do not need to participate in the interaction at the same time;
- Synchronization: service users may asynchronously receive notification messages, and service providers are not blocked while producing information.

The realization of the publish-subscribe interaction pattern typically requires the introduction of a mediator party (called here *mediator*) such as the event channel of the CORBA notification services [89]. The mediator plays the role of a service provider, while both publishers and subscribers play the role of service users. Subscribers have the ability to express their interest in a piece of information (called here *event*), or a pattern of events, and are subsequently notified of any event, generated by a publisher that matches their registered interest. *Figure 2-12* illustrates the publish-subscribe mechanism.

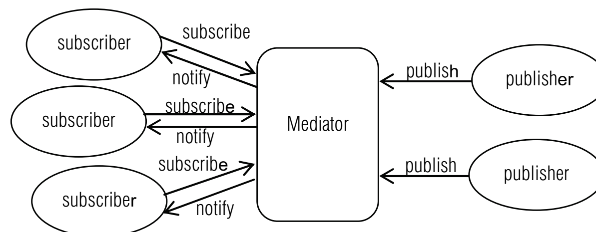



*Figure 2-12* Publish-subscribe interaction pattern

Publishers initiate one-way messages to convey the desire to publish event notifications with the mediator. Subscribers initiate a subscribe message indicating the interest in receiving notifications about particular events or event patterns. Variations of the publish-subscribe mechanism allow subscribers specifying interest in event notifications based on, for example, topic, content or type. Topic-based publish-subscribe allows participants to publish event notifications and subscribe to particular topics or subjects,

which are identified by keywords. Content-based publish-subscribe allows the participants to publish and subscribe to event notifications based on the actual contents of the event notifications. Finally, the type-based mechanism allows participants to publish and subscribe to event notifications according to event notification types (structures).

When a subscriber's interest is matched against a publisher's event notification, the mediator initiates a message to convey a notify message to the interested subscriber.

## 2.4 Context-Aware Service-Oriented Platforms

### 2.4.1 Motivation

Developing context-aware applications is costly and time consuming. An application may use (combinations of) various types of context information to provide its service, such as the user's location, temperature, proximity, and activity. In addition, applications may require complex computations to reason about context, for example, for deriving whether the user is in a meeting based on his/her current location and his/her proximity to certain colleagues.

These requirements demand the use of (among others): (i) complex and costly sensor technology to acquire context; (ii) communication infrastructure to propagate context from sensors embedded in the user's environment to other application components that are physically distributed; and (iii) computational power to undertake context processing tasks that are complex and resource intensive.

In addition, the dynamic nature of context-aware applications, and the increasing integration of these applications into our daily tasks in a variety of domains (e.g., home, work and leisure), generate rapid changes in the requirements for the technology to support these applications. Although it is not possible to fully predict these changes, the supporting technology can be designed such that it can be dynamically configured to match changing requirements, preferably at system runtime. This calls for a high level of application flexibility, which increases development costs and complexity.

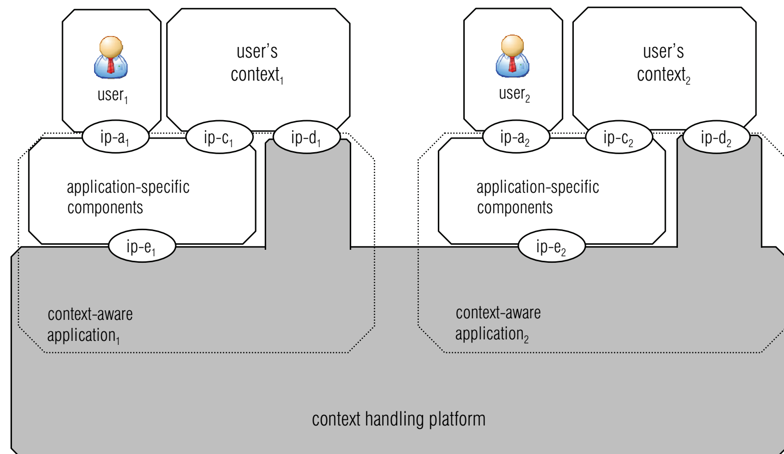
For all these reasons, it is not cost-effective to build each individual context-aware application from scratch. It is also too complex for each individual application to capture and process context information just for its own use. Consequently, context aware applications should be built as families of applications.

### 2.4.2 Context handling platform

We observe that similar functions are repeatedly and, therefore, inefficiently implemented in different context-aware applications or family of applications. For example, a similar mechanism used to capture location information in a tourist application is used in an epileptic healthcare application. These applications would greatly benefit from sharing the same mechanism used to capture location information. Similarly, a number of commonly used functions can be made available for *reuse* to various context-aware applications, by means of *generic services,* which are offered by *generic components*. We call a coherent and self-contained set of generic services to support context-aware application development a *Context Handling Platform*.

In this thesis we aim at coping with the complexity and cost-effectiveness of building context-aware applications by means of a shared *Context Handling Platform*. *Figure 2-13* illustrates our context handling platform (in grey) offering support to two distinct context-aware applications. Context-aware applications use generic services offered by the platform and also implement specific services, which are offered by *application-specific components*. Specific services typically consist of application specific functions that are not worth generalizing, since they are too specific, and therefore not used by other applications. In Chapter 4 we discuss the context handling platform in detail.

*Figure 2-13* Context handling platform offering support to two context aware applications




Users interact with application-specific components through interaction points of type ip-a. Interaction points of type ip-c and ip-d enable the user's context to interact with application-specific components and with the platform, respectively. Interaction points of type ip-c represent the mechanisms used to capture context that are application-specific and cannot be shared. For example, a medical application that measures the

oxygen level in the patient's blood, is normally not allowed to share this information with the platform or with other applications.

Analogously, interaction points of type ip-d represent the mechanisms used to capture context information that may be shared by various applications via the platform. An example is a GPS device that provides location information of a user, which can be relevant to the different applications that might be serving that user. Applications greatly benefit from sharing the mechanism for capturing context, also because it reduces the number of sensors that have to be embedded in the user's context, which reduces the feeling of intrusiveness generated by context-aware applications.

Interaction points of type ip-e enable interactions between application-specific components and the platform. This allows, for example, application-specific services to make use of shared context information.

### 2.4.3 Requirements for building the platform

***Flexibility and extensibility***

The support to context-aware applications by a shared platform should comprise reusable context acquiring, processing and management services. Such services may be based on existing mechanisms that are already deployed, but it should also be possible to dynamically add new services or mechanisms to the platform that will evolve in the future. In particular, the platform may have special components or services that can take application-specified behaviours or procedures as input in order to carry out application-specific context and situational reasoning mechanisms and control actions. In this way application functionality is delegated to the platform, reducing application development effort, time and, therefore, costs. This allows application developers to focus on their core business, instead of being bothered with application realization details. This calls for a high level of flexibility and extensibility.

*Figure 2-14* depicts the context handling platform serving two distinct context-aware applications. A context-aware application is composed of application-specific components, and by platform components. Application-specific components may invoke the platform services, through interaction points of type ip-e, in two ways: by delegating application-specific behaviours to the platform, which in turn will process this particular behaviour on behalf of the application; and by traditionally invoking services, i.e. offering typed parameters. In traditional services, the behaviour of the service itself is defined at service design-time. Conversely, in flexible services, the runtime behaviour of the service depends on the

particular application behaviour being provided as input, and cannot be defined completely at design-time.

*Figure 2-14* Platform flexibility

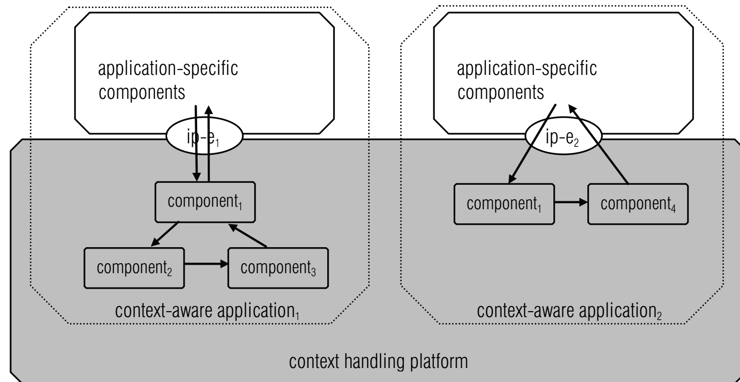



*Figure 2-14* depicts two occurrences of $component_1$ offering flexible services to applications 1 and 2. The behaviour exhibited by $component_1$ upon receiving inputs from application-specific components depends on the nature of the input. Since applications 1 and 2 offer distinct inputs, $component_1$ behaves differently. Similarly, if application 1 offers a different input other than the one depicted in *Figure 2-14*, $component_1$ will take up yet a different configuration of components, and will behave differently. In such flexible services, infinite configurations of components and services are possible.

Our context handling platform should be flexible in order to support flexible services. In addition, the platform should be extensible, so that components and services (flexible or not) can be incorporated on demand in order to cope with newly created applications, which require tackling new introduced requirements.

***Context and situation reasoning***

Context-aware applications are adaptive by nature; therefore, the support platform should be able to carry out adaptation on behalf of applications. Adaptation requires handling contexts and situations. In order to support adaptation, platforms should be able to bridge the gap between context captured by sensors and information that are of particular interest to applications. Several layers of context information processing may be required to bridge this gap.

Different applications may be interested in the same context information, and they may also be interested in context information produced in the different phases of context information processing. Therefore, particular context information should not be dedicated to a single application, but rather be available to a set of collaborative

applications, in order to promote exchange of context, which enriches the context-awareness. In addition, context information produced in the different phases of processing should be also available to these applications.

*Figure 2-15* depicts a graph of context processing components, which are called here *context sources (CS)*. This graph shows a particular configuration of context sources, which can vary depending on the context and situation reasoning activities required. In this example, context information provided through interaction points of type ip-d is gathered by components $CS_1$ and $CS_5$, in applications 1 and 2, respectively. Several layers of context sources are depicted ($CS_1$...$CS_n$), each performing a particular piece of context or situation reasoning. $Application_1$ specific components require context information provided by $CS_4$, which in turn gathers context information produced by $CS_3$, and so forth. $Application_1$ could also gather context information directly from other context sources, such as $CS_1$ or $CS_2$. $Application_1$ collaborates with $application_2$ by sharing the context information produced by $CS_n$.

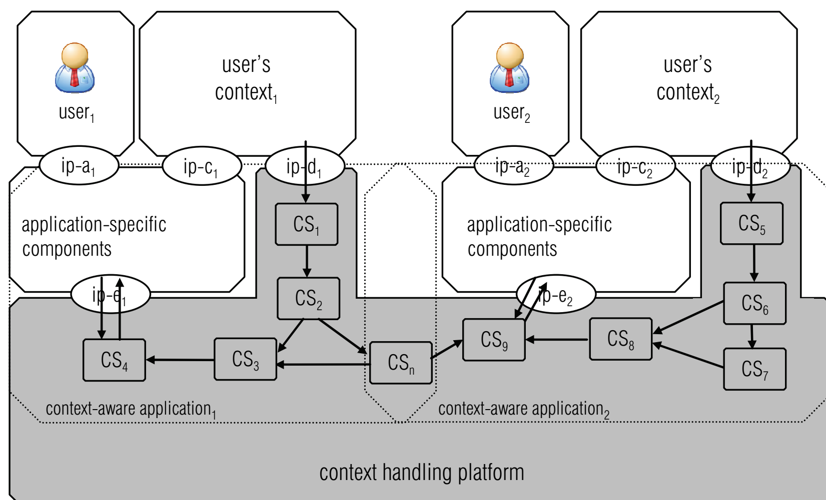



*Figure 2-15* Context and situation reasoning

In order to support context and situation reasoning, context information may be aggregated from different context sources using various composition mechanisms. The platform should support composition of context sources, where the composition itself may be defined at runtime by the application developers. *Figure 2-15* illustrates a particular composition of context sources in which $CS_4$ and $CS_9$ deliver context information as required by the applications. This composition may differ (at runtime) to meet applications changing requirements.

***Distribution***

The platform should also be highly scalable. The number of context sources and context-aware applications may be potentially large and will certainly grow in the near future with further developments of sensor networks and ubiquitous computing devices. At the same time, the amount of context information to be handled by the platform will increase and new context-aware applications may be developed (e.g. in the healthcare domain) that require high volumes of context-related information (e.g. biosignals). It should be possible to support increased numbers and volumes by adding capacity to the platform without changing or disrupting the platform's operation.

In order to allow reuse, scalability and reliability of context and situation reasoning, the context processing phases should be distributed over various context sources. It should be possible to balance context information processing among context sources, and to include or remove context sources at platform runtime.

***Mobility***

Context-aware applications as well as context sources may be mobile (running on a mobile device and attached to mobile objects, respectively), and therefore connections may not rely on computer nodes known in advance. Mobility is an important characteristic that requires explicit consideration from the platform. Different qualities for data transfer and different policies for accessing information and using resources may exist in different environments that an application or context source may experience during a single session. The platform should as much as possible shield the applications from the mechanisms that are necessary to deal with such heterogeneity.

***Rapid development and deployment of applications***

The development and deployment time of using the platform should be reduced with respect to the time required to develop and deploy an application without the support of the platform. Reduction in application development time, costs and efforts should compensate for the time, costs and efforts required to (i) learn how to use the platform, and (ii) install and maintain the platform.

In order to decrease the learning curve, and therefore shorten the development time, platforms should offer application developers mechanisms for deployment and configuration of applications and components which are *easy to use*. These mechanisms should not require much programming and configuration efforts.

### 2.4.4 Classification of services

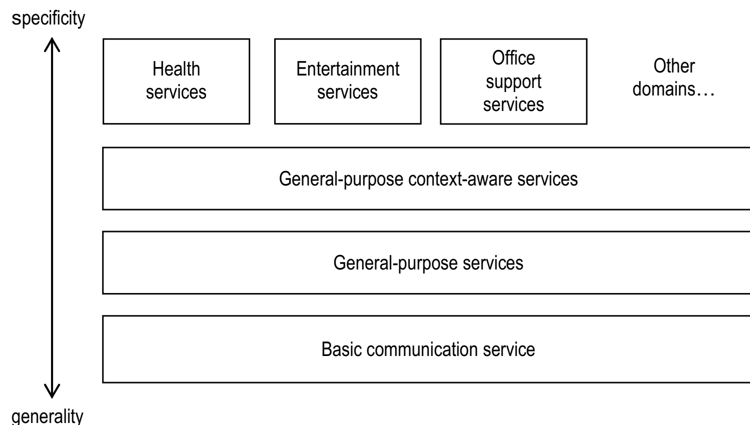
We classify the services provided by a context handling platform according to their degree of generality as:

- General-purpose services: can be used in any application and any application domain. Examples are a directory service, a naming service and an event service;
- General context-aware services: can be used in any context-aware application. Examples are a location discovery service, and an end-user's device monitoring service;
- Domain-specific services: can be used in any application in a specific application domain. A patient record service is an example of domain specific service in the health domain.
- Domain-specific context-aware services: can be used in any context-aware application in a specific application domain. Examples of context-aware services specific for the health domain are a tele-monitoring service and a tele-treatment service.

In addition to these services, we also have the communication services provided by the platform, which enable the components supporting the services to interact. In this thesis, we are mainly interested in general context-aware services. We regard the context handling platform as the particular set of general context-aware services.

This service classification resembles the general purpose (horizontal) and domain-specific (vertical) classification of services defined by the OMG in the Object Management Architecture (OMA) [86]. In our platform we can still distinguish context-aware services as an orthogonal class of services with respect to the services in application domains like health, office and entertainment. *Figure 2-16* shows our classification of services.

*Figure 2-16* Service classification




The service classification depicted in *Figure 2-16* does not imply any specific layering, i.e. components implementing these services are free to invoke

each other's services in any specific way. In practise we observe that components implementing more specific services tend to make use of more generic services.

### 2.4.5 Middleware

The use of a service-oriented approach in the development of our context handling platform architecture implies the introduction of a middleware layer to support the interactions between service providers (application and platform components). For example, the middleware platform can relieve the application designer from explicitly addressing some common tasks distributed applications perform, such as the handling of the reliability of communication, the correlation of requests and responses, the registration, location and activation of application components, the encoding and decoding of messages, the use of a transport protocol, and the replication of application components [2].

*Figure 2-17* shows how application and platform components organised according to the service-oriented approach interact on top of a middleware platform. A benefit of the service-oriented approach is that the designer can abstract from the mechanisms necessary to implement the distribution of application components, which should be considered separately in the scope of the internal structure of the middleware platform.

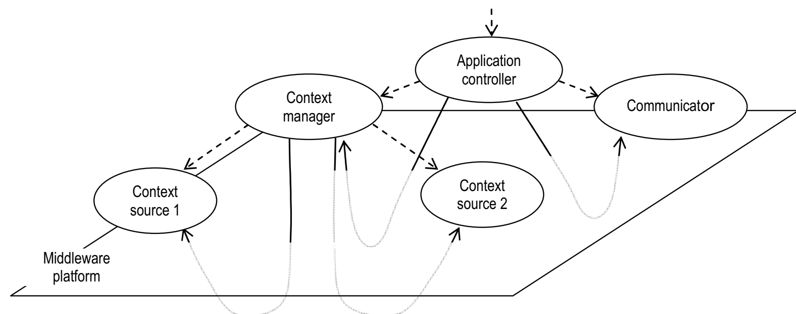



*Figure 2-17* Interactions between application-specific and platform components supported by a middleware platform

The view of the application structure that hides the mechanisms that support the interactions between the application components corresponds to the ODP-RM notion of computational viewpoint [61].

By providing mechanisms that hide from application developers the complexity of implementing component distribution and interactions, the middleware platform is said to offer *transparencies* [121]. Examples of transparencies include *transaction transparency*, which masks coordination of activities between components in order to achieve consistency, and l*ocation, migration* and *relocation transparencies,* which allow the search and invocation of services independently from their location, and from any relocation or migration of components. According to our classification of services

presented in 2.4.4, these are transparencies offered by means of general-purpose services, i.e. can be used by any application and any application domain.

Similarly, other types of middleware may offer more specific services, such as general context-aware services, which serve applications in the context-aware domain. One could use a combination of middleware platforms, such that a specific-services platform uses a generic-services platform, as depicted in *Figure 2-18*. The context-aware middleware platform offers transparencies by means context-aware services. This middleware platform shields application components from dealing with details of, for example, the distribution of context and situation reasoning algorithms, or detecting and reacting upon context information changes, and so forth. In this sense, our context handling platform can be seen as a context-aware middleware platform.

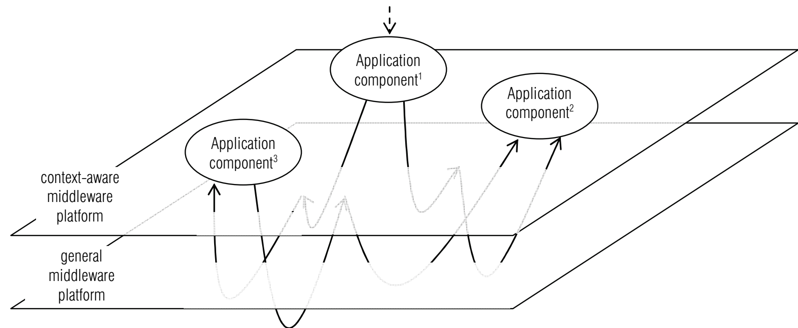



*Figure 2-18* Two layers of middleware platforms

In Chapter 4 we discuss our context handling platform in detail.

Chapter 3

# State-of-the-Art

This chapter presents the state-of-the-art in context-awareness. We discuss related work using the concepts and terminology we have introduced in Chapter 2. Our discussion focuses on three main branches of research:

- *Context modelling*: regards modelling techniques to represent and reason about context and context information;
- *Middleware and platforms*: regards reusable building blocks that facilitate the development and deployment of context-aware applications, including conceptual frameworks to support context-aware application development;
- *Applications*: includes innovative context-aware applications, regardless the use of middleware, platforms or modelling techniques. Typically these applications introduce novel context sensor mechanisms.

The remainder of this chapter is organized as follows: section 3.1 presents examples of context modelling approaches; section 3.2 discusses related work with focus on platforms and middleware, and section 3.3 provides links to relevant initiatives focusing on applications.

## 3.1 Context Modelling

Several context modelling approaches have been proposed in the literature. In the beginning, context models were based on simple data structures, such as key-value pairs [113, 115]. Over time, context-aware applications became more complex and interconnected, while demanding more sophisticated modelling techniques that could provide more complex data structuring.

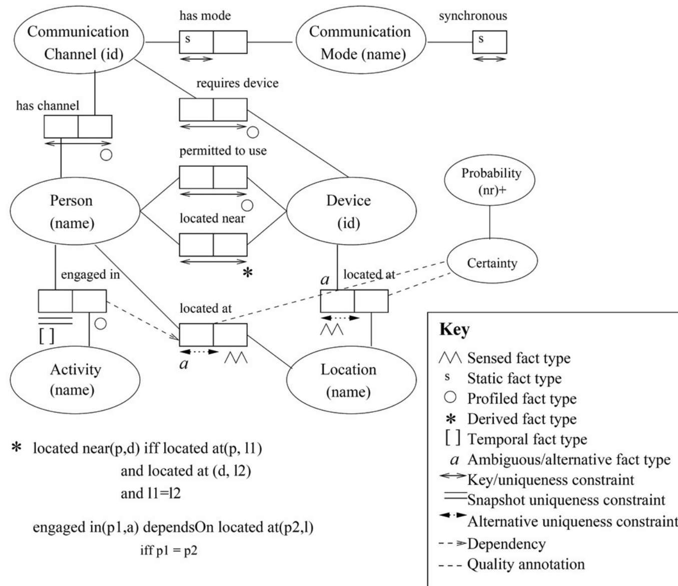
Currently, most context modelling approaches are based on ontology techniques [17, 103, 117]. Fewer models are based on the *Unified Modelling Language (UML)* [13, 45] and the *Object-Role Modelling (ORM) [56]*. The following sections enumerate relevant related initiatives.

### 3.1.1 Context modelling language (CML)

The work presented in [56, 57] describes the Context Modelling Language (CML), which is the most relevant context modelling approach from those based on ORM. CML is a formal, graphics-oriented context-modelling technique that uses the *fact* as the basic modelling concept, and the *situation abstraction* to leverage on facts in order to represent higher level context information. A fact represents any information that is truth in the system, and a fact type refers to the information type. Facts and fact types are defined in an ORM model.

CML extends ORM to allow fact types to be categorized according to their persistence and source, either as *static* (facts remain unchanged while the entities they describe persist), or as *dynamic* (facts dynamically change to reflect changes in the real world). Dynamic facts are further categorized into *profiled*, *sensed* or *derived* types. This approach also includes the notion of history facts, which associates a given fact to a time period in which the fact holds. *Figure 3-1* depicts a context model for a context-aware communication application based on CML.

*Figure 3-1* Fragment of a context model using CML




This model represents users (Person), users' activities (Activity), devices (Device), location of users and devices, communication channels (CommunicationChannel) and communication modes (CommunicationMode). Ellipses represent object types, while each box denotes a role played by an object type within a fact type. User activities are associated with a temporal fact, which allows activities to be recorded over time. Locations of both

users and devices are represented as “sensed” information, and this information has a certainty value associated to it. This certainty is calculated based on a probability estimate of that information to be correct.

In order to allow application designers to represent higher-level types of context (if compared to facts), this approach allows the definition of situations. Situations are defined by constraints on context facts expressed using a variant of predicate logic, and can be combined to define more complex situations. For example, the following expression defines a situation in which the user is able (and allowed) to use a communication channel:

CanUseChannel (person; channel): ∀device • RequiresDevice [channel; device] • LocatedNear [person; device] ^ PermittedToUse [person; device].

This expression defines that a person can use the communication channel (CanUseChannel (person; channel)) when the device required to use this communication channel (RequiresDevice[channel; device]) is close to the user (LocatedNear [person; device]) and the user is permitted to use this device (PermittedToUse [person; device]).

### 3.1.2 Standard ontology for ubiquitous and pervasive applications (SOUPA)

SOUPA [16] is a proposal for a shared, standard ontology for supporting ubiquitous and pervasive computing applications, based on the Web Ontology Language (OWL) [74]. Part of the SOUPA vocabularies are adopted from other ontologies, such as the Friend-Of-A-Friend ontology (FOAF) [12, 98], DAMLTime and the entry sub-ontology of time [100], the spatial ontologies in OpenCyc [70], Regional Connection Calculus (RCC) [109], COBRA-ONT [15], MoGATU BDI ontology [97], and the Rei policy ontology [66].

SOUPA consists of two sets of ontologies: SOUPA Core and SOUPA Extension. The core ontologies attempt to define the vocabularies which are common across different pervasive applications. This set of ontologies consists of vocabularies for expressing concepts that are associated with person, agent, belief-desire-intention (BDI), action, policy, time, space, and event.

The extension ontologies extend the core ontology, and aim at providing additional vocabularies which are common to specific applications. This set of ontologies is defined with two purposes: (i) define an extended set of vocabularies for supporting specific pervasive application domains, and (ii) demonstrate how to define new ontologies by extending the SOUPA Core ontologies. *Figure 3-2* depicts a fragment of the SOUPA core ontology, which describes a Spatial ontology. This fragment of the SOUPA model represents a SpatialThing concept, which is associated with a three-

dimensioned location coordinate data type. A SpatialThing represents a *thing* that occupies space. The GeographicalSpace concept inherits from a SpatialThing. The associations spatiallySubsumedBy and spatiallySubsumes characterizes the contained and the contain relations, respectively.

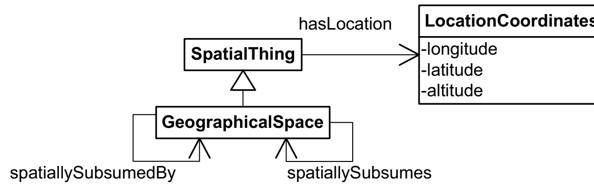



*Figure 3-2* Fragment of the SOUPA ontology regarding Spatial concepts [51]

### 3.1.3 Context ontology language (CoOL)

The Context Ontology Language [117] aims at enabling context-awareness and contextual interoperability in the service discovery and execution phases in service-oriented architectures. CoOL is derived from an Aspect-Scale-Context (ASC) model, which introduces the concepts of *aspect*, *scale, entity, context information, context* and *situation*. *Context information* is defined in CoOL as any information that can be used to characterize the *state* of an entity concerning a specific *aspect*. An *entity* is an object, such as a person, or a place. An *aspect* defines a particular classification of context, which may be characterized by one or more related dimensions called *scales*. A *context* is the set of all context information characterizing the entities relevant for a specific task in their relevant aspects. A *situation* is defined as a set of all known context information.

The CoOL model is a projection of the ASC model into three well-known ontology languages: OWL and DAML+OIL [74], and F-Logic [67]. *Figure 3-3* depicts the Aspect-Scale-Context (ASC) model. This model is actually a metamodel upon which context-models should be based. It defines that an aspect should have a default scale associated to it, and may have a set of scales associated to it. A scale is (i) associated with one or more aspects, (ii) constructed by context information, and (iii) associated with a set of concepts related to possible operations supported by this scale.

Context information is associated with a scale, which defines the range of valid instances of that context information type. A context information type may also be associated with other types of context information, which describe *meta* information about that type, and are called *meta context information types*.

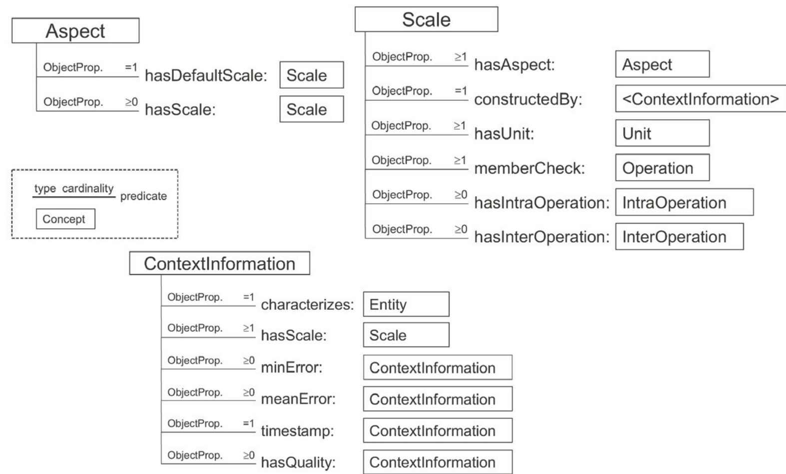



*Figure 3-3* The Aspect-Scale-Context (ASC) model

An example of aspect is GeographicCoordinateAspect that may have two scales, namely WGS84Scale and GaussKruegerScale. A valid context information instance would be an object, instance of one of these scales. This way, every context information type has an associated scale, which defines the range of valid instances of that context information type.

Situations are implicitly defined using a filtering mechanism based on F-Logic. For example, it is possible to define queries, such as "get all entities where the current state is *near*, with respect to the aspect *place*", which filters the context information by specifying conditions (near) on the aspect of interest (place). The corresponding F-Logic query would be:

FORALL E,C,S,V <- C:"urn:cool"#ContextInformation AND C["urn:cool"#characterizes->E] AND C["urn:cool"#hasScale->>S] AND S["urn:cool"#hasAspect->>"urn:aspects"#Place] AND C["urn:cool"#hasValue->V] AND equal(V,"urn:dist"#Near).

Where E states for entity, C for context information, S for Scale and V for value.

### 3.1.4 Context ontology (CONON)

The Context Ontology (CONON) [53] proposes a general context model (called an "upper level ontology") that supports domain-specific specializations. The CONON ontologies are serialized using OWL-DL, which uses description logics to define reasoning and inference rules. Situations are explicitly defined using description logics. *Figure 3-4* depicts a fragment of the CONON upper level ontology.

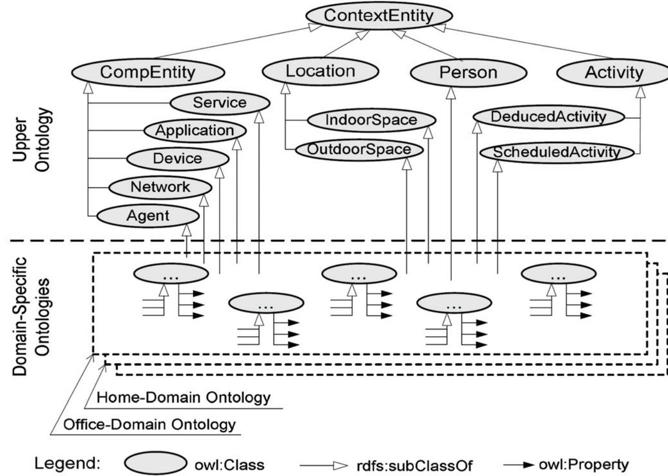



*Figure 3-4* The CONON upper ontology

The context model assumes a set of abstract concepts for the foundations of the ontology. These concepts describe a physical or a conceptual object including Person, Activity, Computational Entity (CompEntity) and Location. The reason for assuming this particular set of concepts has not been reported in the documentation we have studied. The upper ontology defines a set of most used concepts, such as service, application, device, network, and so forth.

Each of these abstract concepts is associated with attributes and relations with other concepts. The built-in OWL property owl:subClassOf allows for hierarchically structuring sub-class concepts, thus providing extensions to add new concepts that are required in a specific domain.

The following examples illustrate how situations can be modelled in CONON:

- Person is sleeping: (?u locatedIn Bedroom) ^ (Bedroom lightLevel LOW) ^ (Bedroom drapeStatus CLOSED) => (?u situation SLEEPING)
- Person is showering: (?u locatedIn Bathroom) ^ (Bathroom doorStatus CLOSED) ^ (WaterHeater status ON) => (?u situation SHOWERING)

The first expression defines that a person is in situation sleeping (?u situation SLEEPING) when he or she is located in the bedroom (?u locatedIn Bedroom), the lights are low in the bedroom (Bedroom lightLevel LOW), and the bedroom door is closed (Bedroom drapeStatus CLOSED). The second expression defines that a person is in situation showering (?u situation SHOWERING) when he or she is located in the bathroom (?u locatedIn Bathroom), the bathroom door is closed (Bathroom doorStatus CLOSED), and the water heater is turned on (WaterHeater status ON).

### 3.1.5 Discussion

Most of the approaches towards context modelling presented here do not explore the benefits of conceptual modelling as the first phase in the design process. Therefore, the distinction between the context concepts and context information is not explicitly discussed. These approaches consider technological issues already in the beginning of the design process, giving computational issues precedence over human understandability.

In addition, these approaches do not consider ontologically well-founded theories to support their modelling choices. For example, the concepts of context and entity are frequently used interchangeably, which does not reflect the fundamental characteristics of these concepts.

We have identified a set of parameters to compare the context modelling approaches presented here. These parameters have been discussed in detail in chapter 2, sections 2.1 and 2.2. Our intention is not to be complete with respect to modelling requirements, but to highlight the lack of some desirable characteristics in current approaches. This discussion forms the motivation for our own context modelling approach, presented in Chapter 5. The following criteria have been used for the comparison:

- *Support for conceptual modelling*: conceptual modelling helps promoting common understanding, problem-solving, and communication among the various stakeholders involved in application development since the beginning of the development process;
- *Support for ontological foundations*: since conceptual modelling focuses on supporting structuring and inferential facilities that are psychologically grounded [9], the adequacy of conceptual models rests on how it contributes to common understanding. Therefore, it is important to justify the modelling decisions with foundational ontologies [4, 6], which are theories based on proven results from conceptual theories in philosophy and cognitive sciences;
- *Separation of context and context information*: context and context information are fundamentally different concepts, and should be treated as such in context modelling;
- *Support for situation reasoning*: in addition to providing proper characterization of the application's universe of discourse, context modelling techniques should offer means to represent particular state-of-affairs of concern, which are often called *situations*;
- *Support for situational temporal aspects*: context-aware application behaviours can be defined in terms of how applications evolve from situation to situation. This requires mechanisms that reason about temporal aspects, such as situation duration, precedence, and overlapping;

– *Support for quality of context information*: context-aware applications actually use context through context information, which is the context measured by sensor mechanisms. These measuring mechanisms may introduce imperfections, which affects the quality of the information (quality of context). Therefore, context information modelling techniques should provide support for modelling the quality of context information. Examples of quality parameters are *accuracy* and *freshness*.
– *Tool Support*: context modelling approaches should offer tool support for (graphically) representing context and situation models. Ideally, these tools should also offer support for brigding the gap between context models and application realizations.

In *Table 3-1* we present a comparative table evaluating the aforementioned context modelling initiatives using the dimensions just presented. We indicate whether the project addresses the respective dimension with the symbol •. The symbols ••, and • demonstrate that the project comprehensively addresses or partially addresses the dimension, respectively. The symbol x indicates that the project does not address the dimension.

*Table 3-1* Evaluation of context modelling approaches.
Legend:
•• = comprehensive support
• = partial support
x = no support

| dimension / project | conceptual modelling | ontological foundations | context and context information | Situation reasoning | situation temporal aspects | quality of context | tool support |
|---|---|---|---|---|---|---|---|
| CML | x | x | x | •• | • | •• | x |
| SOUPA | • | x | x | x | x | • | • |
| CoOL | • | x | x | •• | x | • | • |
| CONON | • | x | x | •• | x | • | • |

SOUPA, CoOL and CONON partially (and implicitly) address conceptual modelling by providing an upper level ontology that represents general concepts, and can be further extended with particular application or domain requirements. However, this (partial) support does not provide the level of abstraction required in the conceptual modelling phase, since the selection of the particular concepts presented in the upper ontology has not been justified from a conceptual modelling point of view, as discussed in chapter 2, section 2.1.2 .

None of the initiatives support or refer to results from ontological foundations in their context modelling approach. Analogously, none of the projects explicitly distinguishes the concepts of context and context information. In CML, in particular, there is no graphical (nor semantic) distinction between the concepts of context and entity, which is inconsistent with the definition of context we have provided.

Except for SOUPA, all initiatives fully support situations by means of logic-based approaches, such as description logics or F-logics. However, although it is possible in most models to represent situations, these models do not allow the definition of temporal relationships amongst situations. CML partially supports temporal aspects of situations by means of defining temporal parameters to the situation, such as situation starting time and ending time. However, temporal relationships between situations are not explicitly supported by CML. CoOL and CONON do not define a suitable notion of time with respect to situations; therefore, it is not possible to define situation duration or any situation temporal relations, such as situation duration or precedence.

Except for CML, all approaches offer tool support for graphically representing context. However, there is no support for graphically representing situations, which are often defined in terms of plain text using some sort of logics. Although CML is based on a graphical notation, CML does not provide tool support for graphically representing both context and situations. We consider tool support as an essential requirement for our context modelling approach.

With respect to the support for quality of context information, CML provides strong support by means of defining various quality of context parameters, such as freshness and accuracy. CONON, CoOL and SOUPA acknowledge the importance of quality of context by partially addressing some quality parameters, such as freshness.

We conclude from our research that there is complete lack of support to conceptual modelling of context in most current approaches, and, therefore, we should address this aspect with particular interest in our context modelling abstractions. We have also seen that many of the related initiatives address situation modelling, but none provides the required level of expressiveness, especially with respect to temporal aspects. In particular, the situation theory presented in CML can be used as an inspiration for our own context modelling approach, since it tackles (to some extend) many of the issues required for modelling context and situations. We have also learned from our research that it is useful to provide metamodels with which application developers can specialize with application-specific concepts. Using a metamodelling approach, application developers are provided with a starting point and guidelines that can facilitate the design process.

## 3.2 Middleware and Platforms

Various platforms and middleware to support context-aware development have been discussed in the literature. We present here some initiatives that we consider most relevant.

### 3.2.1 The Context Toolkit

The context toolkit [24] provides a set of abstractions that can be used to implement reusable components for context sensing and interpretation. This work has pioneered in proposing generic support for context-aware application development by means of a conceptual framework.

The *widget* abstraction represents a component that is responsible for acquiring context information directly from sensors, proving a shielding mechanism to allow uniform usage of sensors, regardless of differences in technology. *Interpreters* combine and interpret context from widgets to provide higher level context information. *Aggregators* combine related context information from different sources (widgets, interpreters or other aggregators) in order to provide compound context information from a single component. Finally, *services* can be used by context-aware applications to invoke actions using actuators, and *discoverers* are the components used by applications to locate suitable widgets, interpreters, aggregators and services.

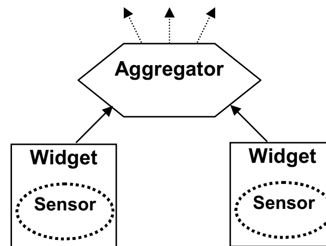



*Figure 3-5* Example configuration of the context toolkit conceptual framework

*Figure 3-5* depicts a particular configuration of the context toolkit, with two widget components and an aggregator component. This aggregator fuses context information provided by these two widgets that encapsulate two sensors. These sensors could provide, for example, the location of a person (by means of a GPS device) and the current schedule of that person. Since these are particular pieces of context information from the same person, the application might be interested in aggregating them, using this particular instance of the Aggregator component. The activities necessary to aggregate context information from different widgets are hard coded in the aggregator component. The toolkit does not provide means to specify and

configure particular aggregation activities at runtime, when the aggregator component is already running.

### 3.2.2 The context broker architecture (CoBrA)

The CoBrA project [16] is an agent-based architecture for supporting the development of context-aware applications. Central to this architecture is an agent called context broker that maintains a shared model of context based on SOUPA (see section 3.1.2) on behalf of a set of collaborating agents, services, and devices using this broker.

The context broker can infer knowledge based on information sensed from the physical sensors and can detect and resolve inconsistent knowledge that often occurs as the result of imperfect sensing. CoBrA also proposes a policy language that allows users to control distribution and notification of user's context information.

Figure 3-6 depicts the CoBrA architecture. A context broker acquires context information from heterogeneous sources, such as smart tag sensors, devices, and gadgets. The broker aims at aggregating context in a model (SOUPA ontology) that is then shared with other collaborative entities.

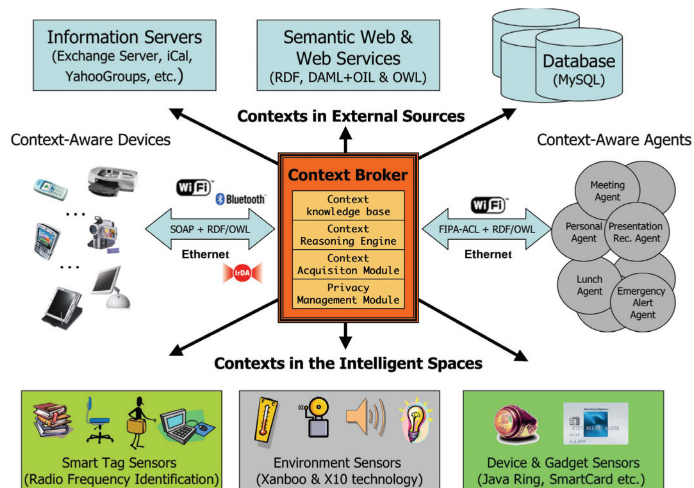



*Figure 3-6* The CoBrA architecture

The context broker consists of four functional components:

- Context Knowledge Base: manages the centralization of the context broker's knowledge, which includes the definitions described in the SOUPA ontology. This component also detects and resolves inconsistencies;
- Context Reasoning Engine: reasons about context information in order to provide higher level context;
- Context Acquisition Module: acquires contextual information from all kinds of context sources and

– Privacy Management Module: aims at protecting privacy by enforcing policies that users have defined to control the sharing and use of their contextual information.

### 3.2.3 Middleware infrastructure to enable active spaces (Gaia)

Gaia [105] is a middleware infrastructure aiming at facilitating the development and deployment of context-aware applications in active spaces. The term *active space* refers to spaces (e.g., office rooms and homes) that are equipped with a large variety of devices, being capable of sensing user actions in order to assist users in different tasks. The Gaia middleware proposes a set of core services for building applications to be developed as loosely coupled components.

*Figure 3-7* The Gaia architecture

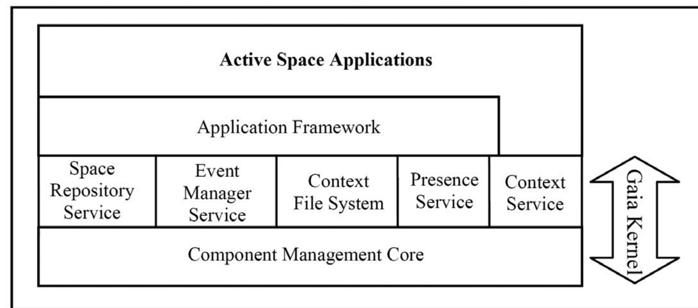



*Figure 3-7* depicts the Gaia architecture. Gaia's five basic services (called Gaia Kernel services) are the Presence Service, Context Service, Context File System, Space Repository Service, and Event Manager Service. These services offer functionality to, respectively, (i) detect people, applications and devices entering and leaving the space, (ii) query and register for context, allowing applications to find and use the proper context information providers, (iii) organize context information in an effective way, (iv) maintain a space repository with hardware and software descriptions, and (v) monitor event sharing among entities in the space.

Gaia's applications use a set of component building blocks, organized as the Gaia Application Framework, to support applications running in an active space. The framework provides support for application mobility, adaptation, and dynamic binding. The Active Space Application layer contains applications and provides functionality to register, manage, and control these applications through the Gaia Kernel services.

### 3.2.4 Reconfigurable context-sensitive middleware (RCSM)

The RCSM [118] aims at facilitating the development and runtime operation of mobile context-aware applications. In this approach, context-

aware application-specific components should implement two separate parts: (i) an interface that encapsulates the application's context sensitivity, such as the list of contexts the application uses, a list of context-triggered actions the application provides, and the rules defining under which context conditions the actions should be triggered; and (ii) the actual implementation of the context-triggered actions that the application must provide. In order to specify application-specific interfaces, RCSM provides the applications with an Interface Definition Language (IDL). Action implementations are not limited to any specific programming language.

IDL interfaces are compiled to produce Adaptive Object Containers (ADC), which are the underlying system to acquire context information. It also communicates with others ADCs in order to activate actions. The communication among ADCs is supported by a context-sensitive object request broker (R-ORB), similar to the (CORBA ORB) as the key mechanism for providing communication transparency. The R-ORB also performs device and service discovery.

### 3.2.5 Pace middleware

The Pace middleware [58] consists of a set of components and tools that can be combined and used by application developers to facilitate the development of context-aware applications. The following components are described by the Pace middleware:

- Context management system: provides mechanisms to perform aggregation and storage of context information. The underlying context modelling approach supported by the Pace middleware is based on CML (see section 3.1.1);
- Preference management system: manages user's preferences in order to trigger actions according to the context and user's preferences;
- Programming toolkit: offers a conceptual model that provides a mechanism for invoking actions by evaluating choices with respect to user preferences and context information;
- Messaging framework: aims at facilitating the communication between the different components of the middleware and context-aware applications. It offers transparencies similar to traditional middleware, such as CORBA;
- Schema compiler toolset: provides a set of tools capable of generating code. For example, it provides SQL scripts to load and remove context model definitions from relational databases, which are used to store context information over time.

### 3.2.6 Discussion

Most of the platforms and middleware that support context-aware application development focus on providing mechanisms for (i) resource and service discovery, (ii) presence of users and devices in smart places, (iii) communication amongst applications and devices in a smart environment, and (iv) transparent context sharing. Few approaches concentrate on context reasoning activities, which combine well-founded context models with components capable of reasoning about context according to these models. The reason for this is that most approaches focus on supporting ubiquity of heterogeneous applications and devices, rather than modelling context. Therefore, ubiquitous computing issues, such as device discovery and ad-hoc communications are sometimes given precedence in these systems over context information handling issues such as context modelling and reasoning.

We have enumerated a number of characteristics that context handling Platforms and middleware should exhibit. These characteristics have been discussed in detail in chapter 2, section 2.4.3. We have chosen these particular dimensions for comparing the developments because of their relevance to this thesis. This discussion forms the motivation for our own context handling platform, which is described in chapter 4. The following dimensions have been identified:

- *Flexibility and extensibility*: platforms and middleware should provide support for flexible components, which are special components or services that can take application-specified behaviours or procedures as input in order to carry out application-specific context and situational reasoning mechanisms and control actions. This allows application functionality to be delegated to the platform (or middleware), reducing application development effort, time and, therefore, costs. In addition, it should be possible to extend platform functionality on demand;
- *Ease of use*: platforms and middleware should offer application developers mechanisms for deployment and configuration of applications and components which are easy to use. These mechanisms should not require much programming and configuration efforts;
- *Support for adaptation*: context-aware applications are adaptive by nature: application behaviours should be adapted to current context conditions. Therefore, platforms (and middleware) should be able to carry-out application-specific adaptation, preferably in a flexible manner;
- *Support for context and situation reasoning*: platforms and middleware to support context-aware application development should provide support for context and situation reasoning. They should be able to bridge the gap between context captured by sensors and context information which are of particular interest to applications. In addition, context

information should be available to applications in the different phases of context information processing;

– *Support for distribution of situation reasoning*: in order to allow reuse, scalability and reliability of context and situation reasoning, the context processing phases mentioned in the above should be distributed over various computing entities (*context sources*). It should be possible to include or remove context sources at platform (or middleware) runtime;
– *Support for runtime composition of context sources*: in order to support context and situation reasoning, context information may be aggregated from different context sources, using various composition mechanisms. The platform (or middleware) should support composition of context sources, where the composition itself may be defined at runtime by the application developers;
– *Support for rapid development and deployment of applications*: the development and deployment time should be reduced with respect to the time required to develop and deploy an application without the support of the platform (or middleware).

In *Table 3-2* we present a comparative table evaluating the aforementioned developments on the light of the dimensions just presented. We indicate whether the development addresses or not the dimension with the symbol •. The indication ••, and • indicate that the development comprehensively addresses or partially addresses the dimension, respectively. The symbol x indicates that the development does not address the given dimension.

*Table 3-2* Evaluation of platform and middleware developments.
Legend:
•• = comprehensive support
• = partial support
x = no support

| dimension<br><br>project | flexibility | ease of use | adaptation | situation reasoning | distribution of situation reasoning | runtime composition of context sources | rapid dev. and dep. |
|---|---|---|---|---|---|---|---|
| The context toolkit | • | • | • | x | x | x | • |
| CoBrA | •• | •• | •• | • | x | x | •• |
| Gaia | • | • | • | x | x | x | • |
| RCSM | •• | •• | •• | x | x | x | •• |
| Pace | •• | • | •• | •• | x | x | •• |

Flexibility and extensibility are important concerns to all developments mentioned here. CoBrA, RCSM and Pace score better in this dimension than the context toolkit and Gaia, since they provide means for applications to express particular behaviours to be delegated to the platform (or middleware). For example, RCSM provides the IDL interface, which allows particular application interests to be deployed on the middleware at runtime.

Flexibility greatly influences the development time (dimension "support for rapid development and deployment of applications"). Applications developed with support of a flexible platform (or middleware) save development time, since they relieve application developers from the task of writing programming code from scratch.

The projects CoBrA and RCSM score well in the ease of use dimension, since they offer flexible mechanisms for application configuration that do not require much programming efforts. The IDL interface in RCSM, and the CoBrA policy language provide application developers with domain-specific languages that facilitate defining context-aware behaviours, as opposed to other general purpose languages. The configuration mechanism proposed by Pace, on the contrary, is based on Java, and does not provide specific support for context-awareness.

Adaptation is also tackled by most of the developments. CoBrA, RCSM and Pace provide extensive support for adaptation. In these systems, it is possible to define in a flexible manner the actions that should be invoked in response to various (not predefined) context change patterns. Pace, in particular, proposes a powerful adaptation framework that also considers user's preferences for service adaptation, in addition to context information.

Context and situation reasoning activities have been explored in CoBrA and Pace. CoBrA uses the SOUPA ontology to support context modelling and reasoning. However, no situation theory has been explicitly applied in CoBrA. Pace uses the CML modelling approach, which supports both context and situation reasoning, as discussed in section 3.1.5.

Given the immaturity of the context-awareness research area when the context toolkit was initially proposed, some of the issues later addressed by more recent approaches have not been tackled in the context toolkit. For example, context modelling issues, and semantic interoperability are not addressed in the toolkit.

We conclude from our state-of-the-art research that none of the initiatives presented here address distribution of context reasoning and composition of context sources at runtime. We have also seen that most approaches offer support for flexibility, but none supports configuration of components based on context and situation specifications. The configuration of components in the platform or middleware should reflect particular requirements for context and situation reasoning. We should consider these aspects when designing our context handling platform.

## 3.3 Applications

Much effort in context-awareness has been spent in building innovative applications, without explicit attention for the use of platform, middleware or modelling. We briefly discuss some projects related to this category. We do not provide an extensive discussion on this, since application development without infrastructural support is not the focus of this thesis.

In the beginning, context-aware applications were limited to location information. A number of office and meeting applications have been developed, where location information was easily captured, since offices and meeting rooms are delimited and controllable areas. Examples of such projects are the Active Badge system [122] and the ParcTab [123]. In the Active Badge systems, users wore a special badge device that could locate them anywhere in the building. Once a user's location was known, phone calls could be forwarded to the closest phones, and past locations could be retrieved.

The ParcTab [123] application is based on palm-sized wireless ParcTab computers and an infrared communication system that links them to each other and to desktop computers through a local area network. The aim is to continuously track users' location inside the office, and offer various location services, such as call forwarding and historic location data.

Several car navigation systems have appeared in the market. These systems provide directions to the driver, based on the car's current location (GPS coordinates), and the desired destination. The Dutch product TomTom [116] is a commercial success in Europe and is already available to the public for more than five years. Similar to car navigation systems are the tourist guide applications [104, 119]. These applications provide the tourist with rich maps of the surroundings, and recommendations of suitable services, considering the user's context and preferences. The WASP project [104] delivered the COMPASS application which provides the user with tourist information and context-aware services, such as finding the closest restaurants, and tourist attractions. For example, a tourist who has an interest in history and architecture is served with information about nearby historical monuments.

Lately, innovative context-aware applications have been proposed in the health domain. Typically these applications detect context information related to body conditions, such as heart beat and blood pressure values in order to help patients that are currently in need. An example is the AWARENESS tele-monitoring application [6], which aims at helping epileptic patients in imminence of an epileptic seizure (see Chapter 1). Other examples of context-aware applications in the health domain can be found in [5, 78].

The MIT Context-Aware Computing Group [20] focuses on building applications in various domains, and the main goal is to demonstrate the usefulness and usability of context-aware applications. Some of the interesting projects are the placeMap [59] and the foodLab[69].

The placeMap project develops user-centred mapping applications. The goal of the project is to provide a mapping mechanism in which users' background, needs, motivations and goals are taken into account to adapt the map accordingly. In this way, mapping applications are not objective, but rather subjective to the user's eye. User-centred maps are inspired by the idea that maps should not present a single view of spatial information, like traditional maps. Our individual mental maps of space focus on relationships between discrete spatial objects. Therefore, our mental maps are quite different from one another's. For example, a person with a car may have a very different understanding of the city than a person who travels by foot and subway. The person travelling by foot has a much more fragmented set of spatial relationships, which are centred on proximity around subway stations.

The foodLab project aims at augmenting the kitchen with devices and sensors that can detect various types of context conditions in order to assist the user in various tasks. For example, one of the topics of research is to equip a spoon with sensors that can measure temperature, acidity, salinity, and viscosity of the food. The goal is to provide information about any food the spoon is in contact with, and to offer suggestions to improve the food.

Chapter 4

# Architectural Patterns and the Context Handling Platform

This chapter presents the three architectural patterns we have proposed to support the development of context-aware platforms and applications. These patterns present solutions for recurring problems associated with managing context information and proactively reacting upon context changes. These problems have been discussed in Chapter 2 as part of the requirements for the platform. In addition, this chapter presents the generic components that integrate the *context handling platform*. The selection of these components is guided by the architectural patterns discussed.

This chapter is organized as follows: section 4.1 introduces architectural patterns as means of documenting design solutions for well-known recurring problems; sections 4.2, 4.3 and 4.4 present the proposed context-aware architectural patterns; section 4.5 discusses our context handling platform architecture in detail; section 4.6 provides an overview of the services offered by the context handling platform; section 4.7 elaborates on the stakeholders involved in the platform development and commercialization; and finally, section 4.8 presents some concluding discussions.

## 4.1 Context-Aware Patterns

Architectural patterns have been proposed in many domains as means of capturing solutions to recurring design problems that arise in specific design situations. They document existing, well-proven design experience, allowing reuse of knowledge gained by experienced practitioners [10]. For example, a software architecture pattern describes a particular recurring design problem and presents a generic scheme for its solutions. The

solution scheme contains components, their responsibilities and relationships.

Patterns for software architectures also exhibit other desirable properties, such as [10]:

- patterns provide a common vocabulary and understanding for design principles;
- they are a means for documenting software architectures;
- they support the construction of software with defined properties;
- they support building complex and heterogeneous software architectures; and
- they help managing software complexity.

We present three architectural patterns that can be applied beneficially in the development of context-aware platforms, namely the *event-control-action pattern*, the *context sources and managers hierarchy pattern* and the *actions pattern*. These patterns present solutions for recurring problems associated with managing context information and proactively reacting upon context changes.

The approach chosen to present these patterns has been inspired by [1], which describes a pattern as a three-part scheme: (i) a situation giving rise to a problem; (ii) the recurring problem arising in that situation and (iii) a proven solution to the problem. Therefore, for each pattern presented here, we discuss (i) an example situation where the problem occurs; (ii) the recurring problem being considered; (iii) the solution scheme for this problem containing structural aspects with components and relationships and dynamic (behavioural) aspects and (iv) the general benefits of applying this pattern.

## 4.2 Event-Control-Action (ECA) Pattern

The *event-control-action architectural pattern* provides a high level structure for systems that proactively react upon context changes. It has been devised in order to decouple context concerns from reaction (communication and service usage) concerns, under control of an application model. An application model defines the behaviour of the application, which may be described by means of, for example, condition rules. In this pattern, context management issues, such as sensing and processing context, are decoupled from issues regarding reacting upon context changes.

### 4.2.1 Example

Suppose our platform needs to provide support for applications in the medical domain. An example of such an application would be a tele-

monitoring application [6] that monitors epileptic patients and provides medical assistance moments before and during an epileptic seizure. Measuring heart rate variability and physical activity, this application can predict future seizures and contact relatives or healthcare professionals automatically. In addition, the patient can be informed moments in advance about the seizure, being able to stop ongoing activities, such as driving a car or holding a knife. The aim of using this system is to provide the patient with both higher levels of safety and independence allowing him to function more normally in society despite his disorder.

In this system scenario, the patient wears a heart monitoring system that collects heart signals along the day. These signals are processed by smart algorithms which are able to detect abnormalities, such as the probability of having an epileptic seizure, within seconds.

Several actions may be taken upon an epileptic seizure: (i) a volunteer, normally an intimate of the patient capable of providing first aid, receives an alarm of a possible seizure, (ii) in case no volunteer is available, healthcare professionals are sent to his location, (iii) patient's bio-signals derived from the monitoring system are streamed to doctors in real time, and (iv) based on the real time information, doctors decide whether the patient needs to be taken to the nearest hospital.

### 4.2.2 Problem

The example presented above imposes challenging requirements to the support platform:

- The platform should offer support for gathering context information, such as the patient's heart rate and blood pressure in order to predict possible epileptic seizures;
- The patient's and volunteers' locations need to be known, and proximity information needs to be derived;
- Full time connectivity with the patient needs to be provided;
- Devices (e.g., mobile phones) of volunteers and doctors need to pass an alarm in case of seizure;
- Real time streaming connections need to be established with the doctor;
- In case of a critical situation, an ambulance needs to take the patient to the nearest hospital.

Implementing such an application within a single business party is not feasible. In fact, this application is realized with the cooperation of several business parties: the location providers, the providers of algorithms to analyze heart rates, the doctors clinic, the hospital, the connectivity providers, and the manufacturers of monitoring devices, among others. The aim of the platform is to guarantee the execution of the application by configuring and coordinating the cooperation of functions distributed

among these business parties. The distribution of responsibilities among these parties and the coordination of distributed functions require agreements on certain architectural patterns.

### 4.2.3 Solution

The event-control-action architectural pattern aims at providing a structural scheme to enable the coordination, configuration and cooperation of distributed functionality within the platform. It divides the tasks of gathering and processing context information from tasks of triggering action in response to context changes, under the control of an application behaviour description. Given the reactive nature of context-aware applications, context-aware application behaviours can be described in terms of reactive rules, such as if <condition> then <actions>. The condition part specifies the situation under which the actions are enabled. Conditions are represented by logical combinations of *events*. An event models the completion of some *happening of interest*, which typically regards a particular change in the users' context. The observation of events is followed by the triggering of actions, under control of condition rules. Events are modelled and observed by one or more *context processor* components.

A *controller* component, empowered with condition rules describing application behaviours, observes the events. In case the condition turns true, an *action performer* component triggers the actions specified in the condition rules. Actions are operations that affect the application behaviour in response to the situation defined in the condition part of the rule. An action can be a simple web services call or a SMS delivery, or it can be a complex composition of services.

### 4.2.4 Structure

The architectural scheme proposed by the ECA pattern consists of three components, namely *context processor*, *controller* and *action performer* components. *Figure 4-1* shows a component diagram of the ECA pattern scheme as it can be applied in context-aware services platforms.

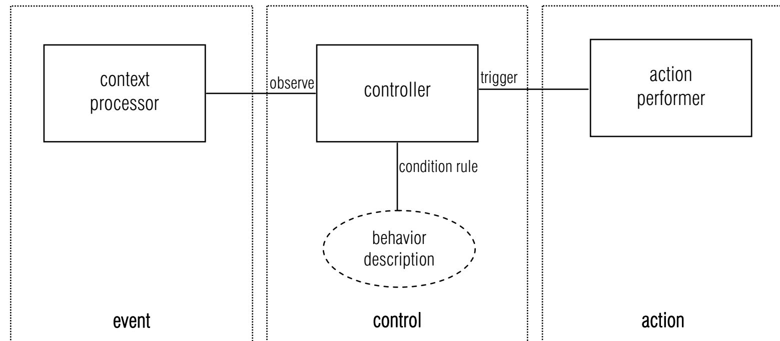



*Figure 4-1* Event-control-action pattern

Context concerns are handled by the context processor component, which generates and observes events. This component depends on the modelling of context information, which is discussed in Chapter 5. The controller component is initially provided with an application behaviour specification, which describes a particular fragment of the application's logic. This behaviour specification determines the observation of context information, and the execution of pertinent actions in response to observed changes in the context. As already mentioned, given the reactive nature of context-aware applications, we suggest a rule-based approach for the specification of context-aware application behaviours. Other specification alternatives are also possible. Provided with condition rules, the controller component observes events from context processors, monitors condition rules, and triggers actions on action performers when the condition is satisfied.

Action concerns, such as decomposition and implementation binding, are addressed by the action performer component. Both the context processor and the action performer components may actually consist of complex composition of components, which characterizes the ECA pattern as a *macro* pattern. This pattern suggests how a context-aware application should be logically structured at the highest level of abstraction. Further decompositions of the components are possible. Context processor, controller and action performer components are further elaborated in the following sections.

### 4.2.5 Dynamics

Consider the example presented earlier in which a possible epileptic seizure is detected and volunteers close to the patient are contacted via SMS. We assume here that, when a possible epileptic seizure is detected, the nearest volunteers are contacted via SMS.

*Figure 4-2* depicts the flow of information between components of the event-control-action pattern. The condition rule (here applied to a patient called John) defined within the controller has the form:

If <event:EpilepticAlarm>
Then <SendSMS(John, closeby(Volunteers, 100))>

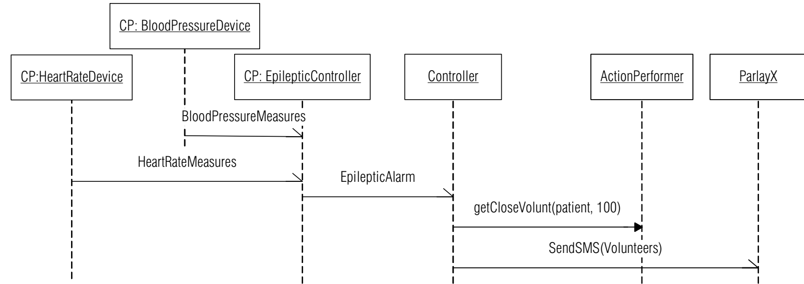



*Figure 4-2* Dynamics of the event-control-action pattern

The controller observes the occurrence of event *EpilepticAlarm*. This event is captured by the component epileptic controller, which is an instance of context processor. Blood pressure and heart rate measures are gathered from other dedicated instances of the context processor. Based on these measures and a complex algorithm, the epileptic controller component is able to predict within seconds that an epileptic seizure is about to happen, and an EpilepticAlarm event is generated as a consequence.

Upon the occurrence of event EpilepticAlarm, the controller triggers the action specified in the condition rule. The action SendSMS(closeby(volunteers, 100)) is a composed action that can be partially resolved and executed by the platform. The inner action closeby (volunteers, 100) may be completely executed within the platform.

The execution of this action requires another cycle of context information gathering on context processors, in order to provide the current location of the patient and his volunteers, and to calculate the proximity of these persons. By invoking the operation getCloseVolunt(patient, 100) with assistance of an internal action performer, the controller is able to obtain the volunteers that are within a radius of 100 meters from the patient. Finally, the controller remotely invokes an action provided by a third-party business provider (e.g., a Parlay X provider [101]) to send SMS alarm messages to the volunteers.

### 4.2.6 Benefits

By applying the classic design principle of separation of concerns, the event-control-action pattern has effectively enabled the distribution of responsibilities in context-aware platforms. Context processor components encapsulate context related concerns, allowing them to be implemented and maintained by different business parties. Actions are decoupled from control and context concerns, permitting them to be developed and operated either within or outside the platform.

Applying such design principles greatly improves the extensibility and flexibility of the platform, since context processors and action components can be developed and deployed on demand. In addition, the definition of application behaviour by means of condition rules allows the dynamic deployment of context-aware applications and permits the configuration of the platform at runtime.

## 4.3 Context Sources and Managers Hierarchy Pattern

The *context sources and managers hierarchy architectural pattern* provides a hierarchical structure for context processor components. This pattern has been devised in order to recursively apply context information processing operations in a hierarchy of context processor components. In this chain of context information processing, the outcome of a context processing unit becomes input for a higher level unit in the hierarchy until the required top-level context information is reached.

### 4.3.1 Example

Suppose we extend the system scenario presented earlier, in which a possible epileptic seizure is predicted. In addition to contacting nearby volunteers, we would like to know whether the patient is driving, in order to send him a personalized alarm, such as "please, stop the car as soon as possible, you may have an epileptic seizure".

### 4.3.2 Problem

Processing context information is challenging. Deducing rich information (e.g., an epileptic alarm) from basic sensor samples (e.g., heart rate and blood pressure measures) may require complex computation. There may be several information processing phases needed before yielding (syntactically and semantically) meaningful context information. Context information processing activities include [29]:

- Sensing: gathering context information from sensor devices. For example, gathering location information (latitude and longitude) from a GPS device;
- Aggregating (or fusion): observing, collecting and composing context information from various context information processing units. For example, collecting location information from various GPS devices;
- Inferring: interpretation of context information in order to derive another type of context information. Interpretation may be performed based on, for example, logic rules, knowledge bases, and model-based

techniques. Inference occurs, for instance, when deriving proximity information from the locations of the objects of concern;
– Predicting: the projection of probable context information of given situations, hence yielding contextual information with a certain degree of uncertainty. We may be able to predict in time the user's location by observing previous movements, trajectory, current location, speed and direction of movements.

The platform should provide mechanisms to distribute context processing activities among multiple components. In addition, it should be able to create compound context information based on various context information sources. Distribution and composition of context information components in a flexible and decoupled fashion require agreements on architectural decisions.

### 4.3.3 Solution

The context sources and managers hierarchy architectural pattern aims at providing a structural schema to enable the distribution and composition of context information processing components. We define two types of context processor components, namely *context source* and *context manager*. Context source components encapsulate single domain sensors, such as blood pressure measuring device or a GPS. Context manager components cover multiple domain context sources, such as the integration of a blood pressure and heart rate measures. Both perform context information processing activities.

### 4.3.4 Structure

The structural schema proposed by this pattern consists of hierarchical chains of context sources and managers, in which the outcome of a context information processing unit may become input for the higher level unit in the hierarchy. The result structure is a *directed acyclic graph*, in which the initial vertexes (nodes) of the graph are always context source components and end vertexes may be either context sources or context managers. The directed edges of the graph represent the (context) information flow between the components. We assume that cooperating context source and manager developers have agreements on the semantics of the information they exchange.

*Figure 4-3* details in the *event* part of *Figure 4-1*. It shows a UML class diagram of the context source and manager hierarchy pattern as it can be applied for context-aware platforms.

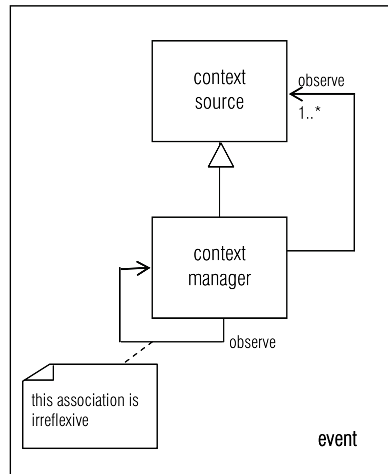



*Figure 4-3* Context sources and managers hierarchy pattern

In the pattern, context managers also play the role of context sources, since a context manager is a source of derived context information. Therefore, context managers inherit the features of context sources, and implement additional functions to handle context information gathering from various context sources and managers. A context manager observes context from one or more context sources and possibly other context managers. The association between the context manager class and itself is irreflexive.

*Figure 4-4* depicts a directed acyclic graph structure, which is an instantiation of the diagram depicted in *Figure 4-3*. CS boxes represent instances of context sources and CM boxes represent instances of context managers.

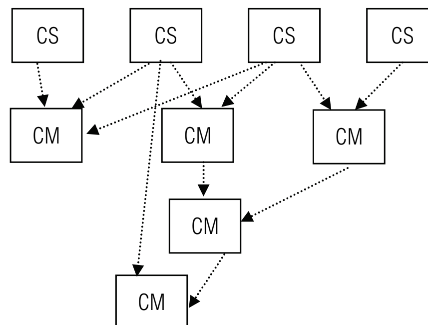



*Figure 4-4* Instance of context sources and managers hierarchy pattern

Within a single context information processing unit (context source or manager), we verify recursive applications of the event-control-action pattern (Section 4.2). Consider the following application condition rule manipulated by controller C1 in *Figure 4-5*:

if <event: (EpilepticAlarm ^ driving)>
then <SendSMS ("please, stop the car as soon as possible, your may have an epileptic seizure")>

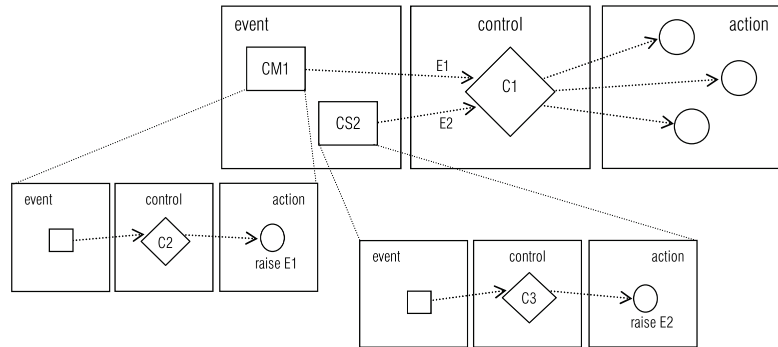



*Figure 4-5* Recursive application of the event-control-action pattern

The event (EpilepticAlarm ^ driving) is a compound event observed on the following components: (i) a context manager component (CM1 in *Figure 4-5*) that detects an epileptic alarm event (E1) and (ii) a context source component (CS2 in *Figure 4-5*) that detects when a patient starts driving[1] (E2). Within the epileptic detector context manager (CM1), the following condition rule[2] is described in controller C2, characterizing the recursive nature of the event-control-action pattern:

if <event:(HeartRate > threshold)>
then <RaiseEvent (EpilepticAlarm)>

Controller C2 observes heart rate measures on a context source component. The action of this rule raises an epileptic alarm event. Within the driving detector context source (CS2), the following condition rule is described in controller C3:

if <event:(userSignalOn)>
then <RaiseEvent (driving)>

The event userSignalOn may be directly set by the patient or automatically sensed by a device embedded in the car that is able to detect his presence.

---

[1] We assume there is a sensor in the patient's car to detect whether he is driving.

[2] For the sake of the example, we simplify the algorithm to detect epileptic seizures by specifying it as the verification of heart rate measures against a threshold value. This algorithm in reality is surely more complex than the simple value comparison given in this condition rule.

### 4.3.5 Dynamics

Consider again the epileptic seizure example discussed in the previous sections. *Figure 4-6* depicts the flow of information between components in the context sources and managers structure at the top most application of the event-control-action pattern. At this level, ControllerC1 observes the occurrence of event (EpilepticAlarm ^ driving), which is generated from CM: EpilepticDetector and CS: DrivingDetector, respectively. When the condition turns true (the alarm has been launched and the patient is driving), the personalized SMS message is sent to the patient.

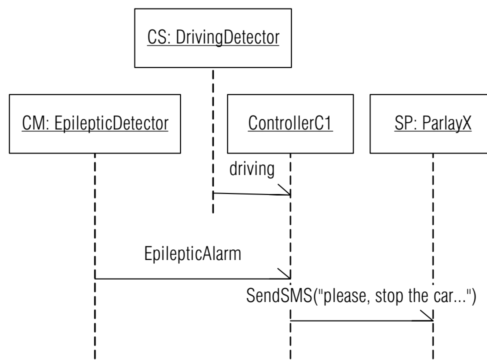



*Figure 4-6* Dynamics of the context sources and managers pattern on the highest level of the event-control-action pattern recursion

In the second recursion level of the event-control-action pattern in *Figure 4-7*, the ControllerC2 observes heart rate measures from a heart device context source. Empowered with algorithms able to detect heart rate abnormality, the controller raises the EpilepticAlarm event when it detects the possibility of an epileptic seizure.

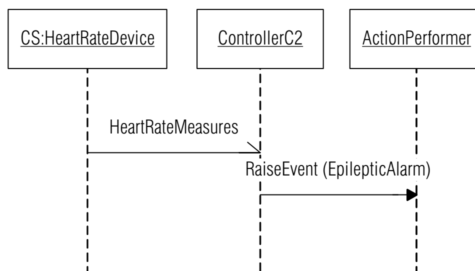



*Figure 4-7* Dynamics of the context sources and managers pattern on the second level of recursion

### 4.3.6 Benefits

The context sources and managers architectural pattern defines a hierarchical structure reference for context source and manager components. This approach has enabled encapsulation and a more effective, flexible and decoupled distribution of context processing activities (sensing, aggregating, inferring and predicting). This attempt improves collaboration

among context information owners and it is an appealing invitation for new parties to join this collaborative network, since collaboration among more partners enables availability of potentially richer context information.

Another important benefit of applying this pattern is that it enables filtering of unnecessary information across the hierarchy of context information processing units. At the lowest level of context information gathering, a great overhead of information flow can be detected but only the relevant information for the application logic is kept and forwarded to the next level of the hierarchy.

## 4.4 Actions Pattern

The *actions architectural pattern* provides a structure of components to support designing and implementing action concerns within the platform. This pattern has been devised in order to decouple action purposes from action implementations and to coordinate composition of actions. An action purpose defines an abstract action intention, while its implementation represents the realization of this intention utilizing specific implementation technologies.

### 4.4.1 Example

Consider the tele-monitoring scenario in which the actions taken upon an epileptic seizure alarm are (i) a warning message is sent to the patient; (ii) his close relatives are called, (iii) volunteers close to the patient are notified of a possible seizure, and (iv) in case no volunteer is available, healthcare professionals are sent to the patient's current location.

### 4.4.2 Problem

Some of the actions presented in the example may be performed independently in parallel, such as (i) sending a warning message to the patient, (ii) calling the relatives and (iii) notifying nearby volunteers. However, the action to call healthcare professionals is only enabled in case notifying nearby volunteers (action (iii)) has not succeeded (for example, no volunteers are momentarily available). This situation characterizes a dependency between actions. In addition, some actions may trigger a sequence of other actions. For instance, to send help from healthcare professionals, it may be necessary to request the patient's medical dossier, to select relevant medication, to check availability of transportation, and so forth.

The platform should provide mechanisms to manage coordination of actions, especially when dependencies exist. In addition, the platform

should support decoupling of an action purpose from its implementations. Although the action "send healthcare professionals" presents a common purpose, its implementations may vary, since the logistics may differ from hospital to hospital. Distribution and coordination of actions in a flexible and decoupled fashion require agreements on architectural decisions.

### 4.4.3 Solution

The actions architectural pattern aims at providing a structural scheme to enable coordination of actions and decoupling of action implementations from action purposes. It involves (i) an *action resolver* component that performs coordination of dependent actions, (ii) an *action provider* component that defines action purposes and (iii) an *action implementor* component that defines action implementations.

An action purpose describes an intention to perform a computation with no indication on how and by whom these computations are implemented. Examples of action purposes are "call relatives" or "send a message". The action implementor component defines various ways of implementing a given action purpose. For example, the action "call relatives" may have various implementations, each defined by a specific telecom provider. Finally, the action resolver component applies techniques to resolve compound actions, which are decomposed into units of action purposes that are indivisible from the platform standpoint.

### 4.4.4 Structure

*Figure 4-8* zooms in on the action part of *Figure 4-1*. It shows a class diagram of the actions pattern as it can be applied for context-aware platforms.

Action resolver and action provider components are special kinds of action performer, being able to perform actions. Therefore, both the action resolver and action provider components inherit the characteristics of the action performer component. The action resolver component performs compound actions, decomposing them into indivisible action purposes, which are further performed separately by the action provider component. Action providers may be communication service providers or (application) service providers. Communication service providers perform communication services, such as a network request, while service providers perform general application-oriented services, implemented either internal or external to the platform, such as an epileptic alarm generation or an SMS delivery, respectively.

An action provider may aggregate various action implementor components, which provide concrete implementations for a given action purpose (represented by implementors A and B in *Figure 4-8*).

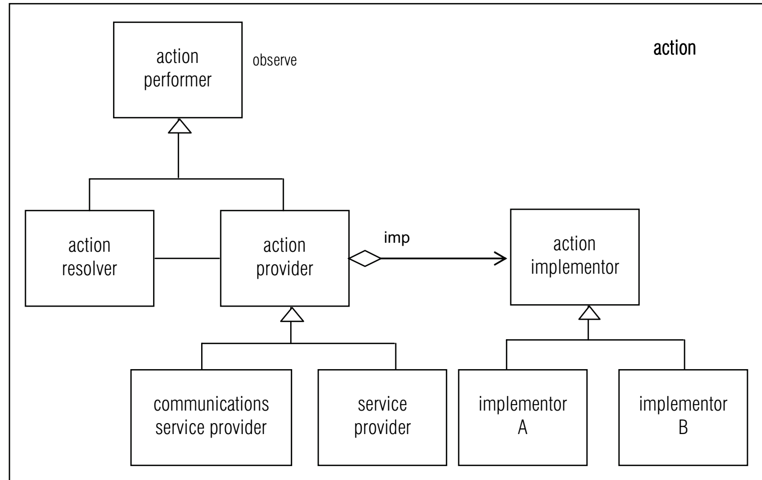



*Figure 4-8* Action pattern structure

### 4.4.5 Dynamics

*Figure 4-9* depicts the flow of information between components of the actions pattern for the scenario presented in Section 4.4.1.

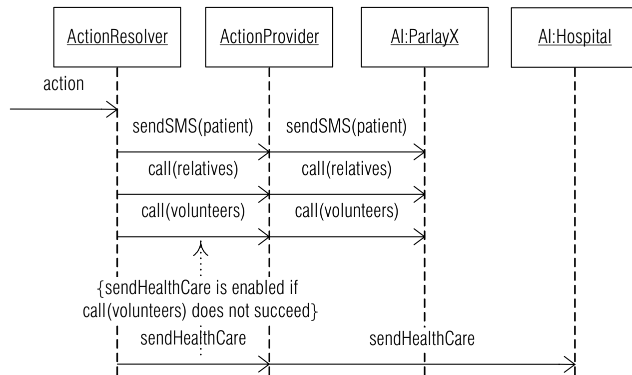



*Figure 4-9* Dynamics of action pattern

The action resolver gets a compound action to decompose. Empowered with techniques to solve composition of services, the action resolver breaks the compound action into indivisible service units, which are then forwarded to the action providers. Each action provider delegates each service unit to the proper concrete action implementation. In our example, send SMS and calling actions are delegated to the ParlayX implementor and the action to send healthcare is delegated to the hospital implementor.

### 4.4.6 Benefits

By defining a structure of action resolvers, providers and implementors, the actions pattern has enabled the coordination of compound actions and the

separation of the abstract action purpose from its implementations. This attempt allows late binding between an action purpose and its implementations, allowing the selection of different implementations at platform runtime. In addition, abstract action purposes and concrete action implementations may be changed and extended independently, with benefits for the dynamic configuration and extensibility of the platform.

## 4.5 The Context Handling Platform

In chapter 2 we have introduced our context handling platform. The aim of this platform is to provide context-aware generic services, i.e. services that support context-aware applications, regardless of application domain. These services can be combined and configured to satisfy specific application requirements.

*Figure 4-10* depicts (i) the context handling platform as we have discussed in chapter 2 (*Figure 4-10 (a)*), and (ii) our focus on the interactions that are relevant for the platform development (*Figure 4-10 (b)*). *Figure 4-10 (a)* shows an example of platform configuration in which two distinct users use the platform services through application-specific components. Both platform and user's contexts participate in the interactions taking place at interaction points of type ip-d.

*Figure 4-10 (b)* focuses on a single application. It abstracts from the entity *user* and the interaction points of type ip-a and ip-c. Since the platform does not participate directly in the interactions taking place in the interaction points of type ip-a and ip-c, these interaction points are not considered in the platform development.

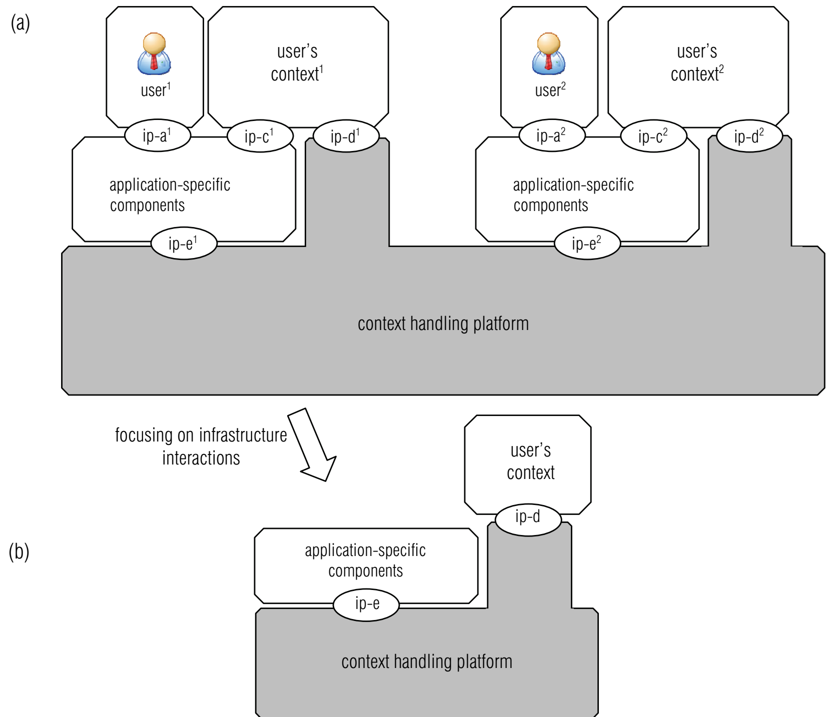
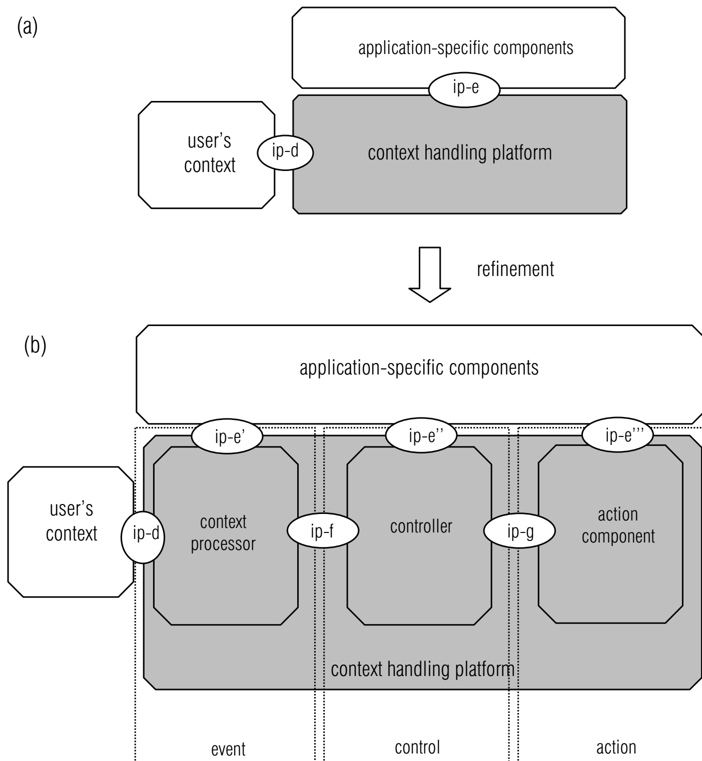



*Figure 4-10* The context handling platform

### 4.5.1 Using the event-control-action pattern

*Figure 4-11 (a)* rearranges the components of *Figure 4-10 (b)* by placing the user's context on the left side of the figure. *Figure 4-11 (b)* depicts a possible refinement of the context handling platform into subcomponents, namely *context processor*, *controller*, and *action components*. The use of the ECA pattern is also represented in *Figure 4-11 (b)* by means of dashed rectangles around the components. The configuration of components shown in this figure demonstrates a particular example in which there is a single occurrence of each of these platform components. Other configurations of components, including multiple component occurrences, are also possible, as can be seen in section 4.5.2.

*Figure 4-11* Context handling platform refinement




In *Figure 4-11 (b)*, the interaction point of type ip-e has been split into three different interaction point types, namely ip-e', ip-e'' and ip-e'''. These interaction points enable application-specific components to interact directly with the different components of the platform. Application-specific behaviours are delegated to the platform through interaction points of type ip-e''. The controller component takes these behaviours as input and configures the rest of the platform to operate properly according to the application-specific requirements. For that, the controller should announce to the context processor components that certain types of context information are needed.

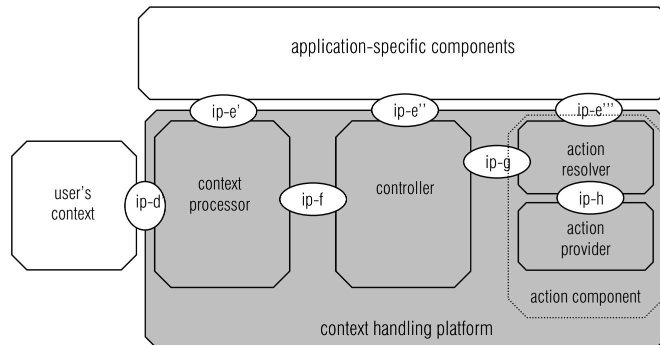
Context processor components are responsible for interacting with the user's context through interaction points of type ip-d. Based on measured information of the user's context, context processors generate context events, which are observed by controller components through interaction points of type ip-f. Application-specific components may interact directly with the context processor components, through interaction points of type ip-e'. This allows application components to access context information, independently of using the controller component.

When the combination of context conditions defined by the application-specific behaviours is met, the controller component triggers the required actions through interaction points of type ip-g. Application-specific components may also interact directly with action components, through interaction points of type ip-e'''. This allows applications to trigger (combinations of) actions, independently of using the controller component.

### 4.5.2 Platform refinements

*Figure 4-12* depicts a refinement of the action component into two subcomponents, namely, *action resolver* and *action provider* components. This refinement follows the solution proposed in the action pattern, as presented in section 4.4.4.

*Figure 4-12* Refinement of the action component




We define two possible types of action provider components that might be capable of performing actions, namely, *external action provider* components and *internal action provider* components. An external action provider component is developed and maintained by a third-party provider, while an internal action provider component is developed and maintained by the platform itself.

Similarly to action components, context processor components may be internal or external. External context processor components are developed and maintained by third-party providers, while internal context processor components are developed and maintained by the platform itself.

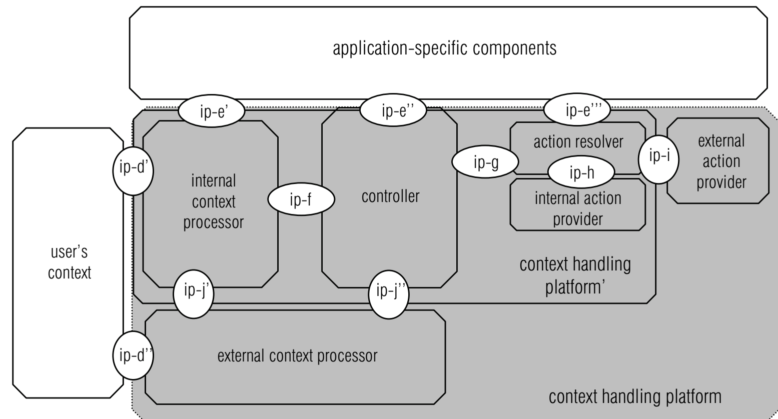
In *Figure 4-13*, we have separated external action providers from internal action providers, and external context processors from internal context processors. Invocations to internal actions are performed through interaction points of type ip-h, and invocations to external actions are performed through interaction points of type ip-i. Internal context processors can be invoked through interaction points of type ip-e', ip-f and ip-j. External context processors can be invoked through interaction points of type ip-j' and ip-j''. The interaction points of type ip-d' and ip-d'' allow

internal and external context processors to capture context conditions from the user's context, respectively.

*Figure 4-13* distinguishes the platform containing external and internal components from the platform containing only internal components, called *context handling platform'*. In this thesis, we are particularly interested in the internal components, which are entirely developed and maintained by the platform. Therefore, we focus on the design and implementation of the context handling platform'. How external components are designed and implemented is not our concern. However, in order for these components to properly operate with the platform, they should follow certain specifications, which are considered in section 4.6.

In the remainder of this thesis we use the term *context handling platform* in place of *context handling platform'* for the sake of clarity and simplicity.

*Figure 4-13* Internal and external components




As discussed in the hierarchy of context sources and managers pattern (section 4.3), a context provider can be refined into context manager and context sources components. *Figure 4-14* depicts a possible refinement of the internal context processor component, in which two context sources and a single context manager collaborate to interpret context information at the right level of abstraction required by the application-specific components and the controller component.

Context source[1] interacts directly with the user's context. This context source could be, for example, an encapsulation of a GPS device to capture the user's location. Context source[2] interacts with an external context processor, which could be, for example, a weather forecast provider. Although it is not shown in the figure, the controller component could interact directly with the context sources, and with the external context processor.

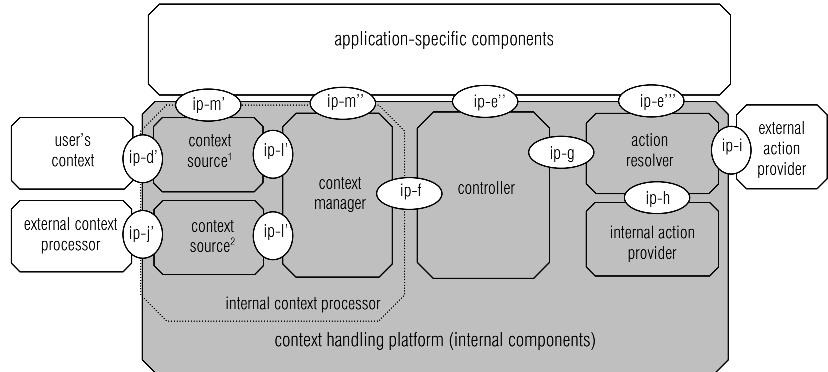



*Figure 4-14* refinement of the internal context processor component

*Figure 4-15* illustrates a different configuration of the platform components, in which two distinct applications (application$^1$ and application$^2$) use the platform. This particular configuration of components uses multiple instances of context processor, controller and action resolver components. The way in which components should be configured depends on the application goals. Different goals may require different configurations of components.

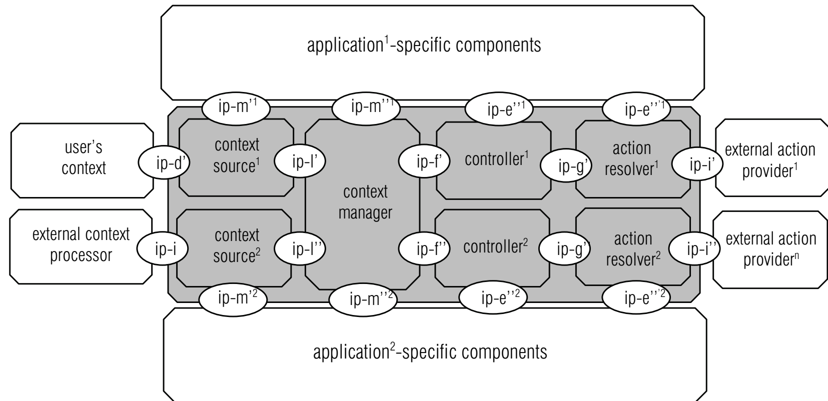



*Figure 4-15* Another example configuration of the components

The components presented in all these figures offer services as in a service-oriented architecture. Therefore, services in our approach are registered and discovered in a service repository (see chapter 2, section 2.3.2). The discovery of services is not depicted in these figures but it implicitly enables interactions between components in the architecture.

## 4.6 Platform Services

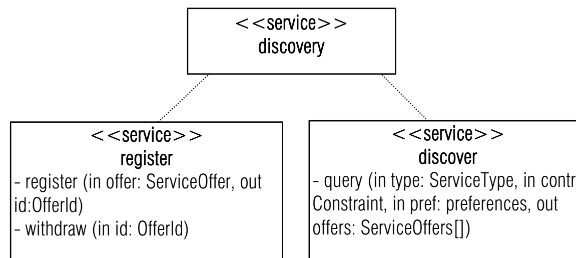
The service-oriented architecture approach implies that components of the platform make use of each other's service in order to support the goal of

the application. In addition, services are the only way to interact with a component, enforcing in this way a discipline in the composition of the application. The following paragraphs define the services offered by the components we have discussed in the previous sections.

### 4.6.1 Discovery services

In section 2.3.2 (see *Figure 2.8*) we have introduced the discovery services, which facilitate the offering and the discovery of instances of services of particular types. The offering of service instances is supported by the *register* service, while the discovery is supported by the *discover* service. These two services together form the *discovery services*, as depicted in *Figure 4-16*. Discovery services are offered by entities called *service directories*.

*Figure 4-16* Discovery services




The *register* service defines two operations: *register* and *withdraw*. The operation *register* allows service providers to register service descriptions with a service directory. Analogously, the operation *withdraw* allows services providers to withdraw service descriptions from a service directory. The *discover* service defines the operation *query*, which allows potential service users to discover services of interest by looking up in service directories.

To register, a service provider gives the service directory a description of a service and the location of an interface where that service is provided. To query, a service user asks the directory for a service having certain characteristics. The directory checks against the service descriptions it holds and responds to the service user with the location of the selected service's interface. The service user is then able to interact with the service.

The following data types are used in *Figure 4-16*: (i) a ServiceOffer represents a description of the service to be included in the service register; (ii) an OfferId is an identification of the service offer; (iii) Constraints define restrictions on the services offers being selected, for example, restrictions on quality of service or any other service properties defined; and (iv) Preferences determine the order in which the selected services should be presented.

### 4.6.2 Context provisioning service

A Context provisioning service facilitates the gathering of context information. This service is supported by context processor components, i.e. context source and context manager components. A context provisioning service may support two types of requests: query-based or notification-based, as already discussed in section 2.3.3. A query-based request triggers a synchronous response while a notification-based request specifies conditions under which the response should be triggered. Example of query-based and notification-based requests are getLocation (user:John) and getLocation (user:John, condition: time=t), respectively. In the first request, the service user immediately gets the current location of user John (assuming this is available). In the second request, the service user gets John's location only when time is t is reached. *Figure 4-17* depicts our context provisioning service.

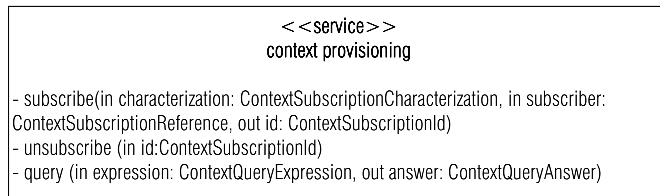



*Figure 4-17* Context provisioning service

Operation subscribe is used to register a notification request, operation unsubscribe is used to withdraw a given notification subscription and operation query is used to select specific context information instances. The specification of languages to define context subscription characterization, context query expression and context query answer is discussed in chapter 6.

Potential users of the context provisioning services are (i) application-specific components, (ii) the controller component and (iii) other context provisioning services.

Context provisioning services may be advertised and discovered using the discovery service. We may define properties of context to be used as constraints to select context provisioning services, such as the quality of context properties accuracy and freshness. The definition of such properties is highly related to the context model discussed in chapter 5.

### 4.6.3 Action service

An *action service* allows users of this service to request the execution of certain actions. This service is offered by the action provider components.

Action implementers provide their action service specifications, which are wrapped in action services supported by the platform. Furthermore, action implementers should register their services in the platform service

directory, setting parameters and properties that should be used in the discovery process. An action provider supports a single standard operation, namely DO (action_name, parameters), which allows an action to be invoked uniformly.

*Figure 4-18* depicts the generation of action wrappers based on an action service specification. This action service is the SendSMS [101] service offered by a telecom provider.

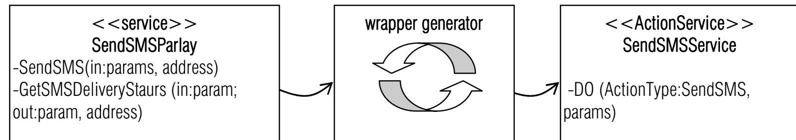



*Figure 4-18* Action services

The SendSMSParlay service specifies two operations, SendSMS and GetSMSDeliveryStatus. This service is wrapped by a service supported by the platform, containing a DO() operation. The wrapper service has pointers to the actual implementations of the operations SendSMSParlay and GetSMSDeliveryStatus. SendSMSParlay service implementers advertise this service in the platform service directory, setting parameters and properties such as costs and location coverage.

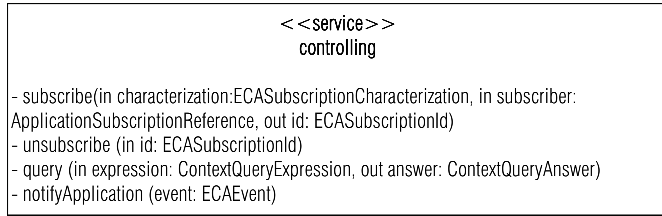
Potential users of the action services are (i) specific application components, (ii) the controller component and (iii) other action services. In order to find action services, action services users should first discover these services with the platform service directory.

### 4.6.4 Controlling service

The *controlling service* allows users of this service to (i) activate Event-Condition-Action (ECA) rules and (ii) query for specific instances of context information.

The controlling service supports the following types of operations: subscribe, unsubscribe, query and notifyApplication. Subscribe is used to activate an ECA rule within the platform; unsubscribe is used to deactivate an ECA rule; query is used to select specific context information and notifyApplication is used to notify application components of the occurrence of ECA events. *Figure 4-19* depicts the controlling service.

*Figure 4-19* Controlling service




The definition of specification languages to define ECA subscription characterization, ECA events, context query expression and context query answer is the topic of chapter 7.

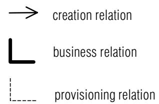
Potential users of the controlling Service are application components that would like to activate ECA rules within the platform. Application components may use this service to get event notifications back from the platform.

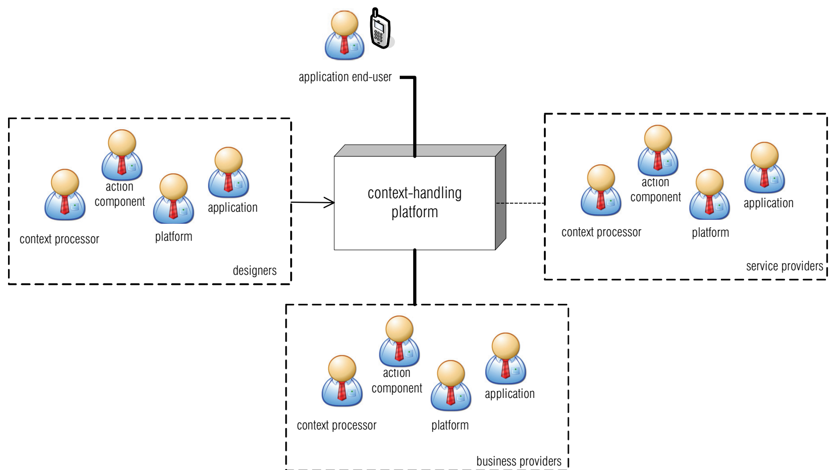
## 4.7 Platform Stakeholders

The context handling platform architecture provides means for various business parties, or stakeholders, to collaborate. Context handling platforms are only realized with the cooperation of various stakeholders, such as the location providers, the providers of context information reasoning algorithms, action providers, and so forth. *Figure 4-20* depicts the types of stakeholders involved in the platform development, service and business provisioning.

*Figure 4-20* Platform stakeholders

Legend:




We have identified three distinct relations between these types of stakeholders and the platform, namely, *creation, business* and *provisioning*

relations. The *creation* relation refers to the development process, which includes design and implementation of components and services. The *business* relation refers to managing the service provisioning and service usage from a business perspective. In the scope of business, service provisioning typically includes marketing, financial and legal issues, and service usage typically refers to the business agreements between the service user and the provider, such as subscription contract and fees. The *provisioning* relation refers to performing service provisioning from a technical point of view, which includes configuration, installation and maintenance of software and hardware artefacts.

In *Figure 4-20*, creation relations are represented by arrowed lines, business relations are represented by thicker not arrowed lines, and provisioning relations are represented by dashed not arrowed lines.

For each type of platform component, there are stakeholders to (i) create that component (creation relation), (ii) provide the service offered by that component from the perspective of business matters (business relation), and (iii) provide the service from a technical point of view (provisioning relation). These stakeholders may coincide, i.e. the same organization may play simultaneously the roles of developer, maintainer or business provider; however, since they are playing different roles, we consider them separately. We discuss each of these stakeholders in the sequel.

### 4.7.1 Platform designer, and business and service providers

Three types of stakeholders are responsible for the development, maintenance and commercialization of the platform, namely *platform designer*, *platform business provider*, and *platform service provider*, respectively. The platform designer is responsible for (i) creating the platform; (ii) defining the component interfaces; (iii) defining basic context models and the guidelines for extending such models; (iv) developing the controller component; and (v) developing internal context processor and action components. In addition, the platform designer provides guidelines to applications developers on how components can be used, configured and composed to meet particular application requirements.

The platform business provider aims at providing the platform services from a business perspective. It tries to maximize the number of applications interested in using the platform by offering business opportunities to application providers, action and context processor providers. The platform enriches its services by allowing third-party action and context processor providers to be incorporated to the platform. This way, a wide variety of services are offered through the platform, which is an appealing invitation to a larger number of end-users, and therefore, to applications. These

business relations are also beneficial to action and context processor providers, since the platform serves as a bridge to reach end-users.

The platform service provider aims at performing the services offered by the platform from a technical point of view. The service provider aims at maintaining high availability of controller, internal context processor and action components. In addition, the service provider is responsible for configuration and installation of such components.

### 4.7.2 Application end-user

Context-aware applications provide value to end-users in a number of ways. For example, end-users may enhance their productivity at work or improve their personal lives by using certain types of context-aware applications. In order to use the applications services, end-users maintain business relations with applications, i.e. they subscribe to applications in various domains such as tourism, health, and office applications.

Application end-users are not expected to interact directly with the platform, but this communication occurs through applications. However, depending on the business model chosen, there may be business relations between end-users and the platform business provider in addition to the business relations between applications business providers and end-users. This could be characterized, for example, by charging end-users for particular platform service usage.

### 4.7.3 Application designer, and business and service providers

Application designer, and business and service providers are the stakeholders responsible for developing, commercializing and maintaining the context-aware applications, respectively. Application designers aim at developing context-aware applications from an engineering perspective. In order to build applications with the support of the platform, application designers should (i) understand the services offered by the platform; (ii) be able to extend the platform context models with application specific concerns; and (iii) be able to combine and configure platform services to meet particular application requirements.

Application business providers aim at providing application services from a business perspective. This stakeholder foresees business opportunities when using the platform, since applications are enriched with a variety of context provisioning and action services, which are offered through the platform. In addition, the platform provides adaptation and attentiveness with the controlling service. The application business provider also controls the business relations towards the platform business provider, and end-users.

The application service provider aims at performing the services offered by the applications. This may include installation, configuration and maintenance of application servers, and the use of mechanisms such as mirrored servers to allow high application availability.

### 4.7.4 Context processor designer, and business and service providers

Context processor designer, business and service providers are the stakeholders involved in the development, commercialization and maintenance of the context processor components, respectively. The context processor designer develops context processor components from an engineering perspective. In order to build context processor components to be available through the platform, the context processor designer should understand and comply with the platform context provisioning services and context models. Context provisioning services should follow the specification discussed in section 4.6. Furthermore, the platform context models should be used by the context processor designer as a blueprint to guide context processing activities, such as interpretation or aggregation.

The context processor business provider aims at providing context provisioning services from a business perspective. Providing context provisioning services through the platform is an opportunity to reach more end-users and to enrich context provisioning services by collaborating with other context provisioning services. Collaboration among more context processor components enables availability of potentially richer context information (see section 4.3.6). The context processor business provider also controls the business relations towards the platform business provider, and end-users, in case such relation exists.

The context processor service provider takes care of performing context processor components by maintaining the software and hardware artefacts necessary to offer such context provisioning services.

### 4.7.5 Action component designer, and business and service providers

Action component designer, business and service providers are the stakeholders involved in the development, commercialization and maintenance of the action components, respectively. The action component designer develops action components from an engineering perspective. In order to build action components to be available through the platform, the context processor designer should understand and follow the platform action service discussed in section 4.6.

The action component business provider aims at providing action services from a business perspective. Providing action services through the platform is an opportunity to reach more end-users and to enrich action services by combining various action services from different providers to

allow composition of various action services. The action component business provider also controls the business relations towards the platform business provider, and end-users, in case such relation exists.

The action component service provider installs, configures and maintains the necessary software and hardware artefacts for performing action services.

## 4.8 Discussion

We have presented in this chapter the architectural patterns that can be beneficially applied in the development of context handling platforms and context-aware applications. We have also discussed our context handling platform, whose architectural design is based on these patterns. Furthermore, we have defined the services that are offered by the context handling platform components.

In the remaining chapters of this thesis, we discuss the components of the platform in more detail. We mainly focus on context processor and the controller component. We do not focus in this thesis on the action pattern, nor on the action components. We use simple examples of actions for the purpose of demonstrating the controller component. Action components and complex action compositions are being studied in parallel efforts [8].

Chapter 5

# Context Modelling

Context modelling refers to the process of preparing context models, which are abstract representations of the context conditions and situations that are relevant in an application's universe of discourse. In order to cope with context modelling requirements, we define a set of context modelling abstractions that are based on conceptual modelling theories [80] and supported by developments in foundational ontologies [51]. These modelling abstractions facilitate the specification of context models that are clearer and easier to understand. Therefore, the context models produced using our conceptual foundations are beneficially applied to promote common understanding and communication between the stakeholders involved in the application development. In addition, by using our context modelling abstractions, application developers are provided with proper conceptual foundations that can be extended and specialized with specific application requirements.

In this chapter we present the modelling abstractions we propose to represent context and situations in our context handling infrastructure. We focus on providing modelling foundations that can be extended with application or domain-specific aspects. This chapter is further organized as follows: section 5.1 identifies relevant characteristics of context and context information; section 5.2 discusses the concepts from foundational ontologies that are used in our modelling approach; section 5.3 presents our own foundation extensions to model context concepts; section 5.4 gives examples of application and domain specific context models, focusing on the different categories of context; section 5.5 elaborates on situation models; section 5.6 discusses context information models, and finally, section 5.7 presents some final remarks.

## 5.1 Characteristics of Context

In chapter 2 we have defined context as [79] "the set of, possibly interrelated, conditions in which an entity exists". According to this definition, context is only meaningful with respect to a thing that exists, called *entity*. The concept of entity is fundamentally different from the concept of context: context is what can be said about an entity in its environment, i.e. context does not exist by itself. The context of an entity may have many constituents, called *context conditions.* Examples of context conditions of a person are the person's location, mental state, and activity. Together, these context conditions form the entity's *context*.

The process of identifying the relevant context consists of determining the "context conditions" of entities in the application's universe of discourse that are relevant for a context-aware application or a family of such applications. The representation of these relevant conditions or circumstances is called a *context model*. We define a context model as a *conceptual model* of context.

Conceptual models are, in the sense of [51, 80], abstract representations of a "given subject domain independent of specific design or technological choices". Therefore, in a conceptual model of context, we abstract from how context is sensed, provided, learned, produced and/or used. The conceptual modelling phase is fundamental for the development of context-aware applications, since this phase produces models that promote "understanding, problem-solving, and communication, among stakeholders about a given subject domain". These models are used as a blueprint for the subsequent phases of a system's development process. Therefore, the quality of context-aware applications depends on the quality of the conceptual context models upon which their development is based.

In our context-aware application development approach, conceptual modelling of context precedes the detailed design of context-aware applications.

## 5.2 Foundational Ontologies

We have observed that conceptual modelling of context shares a great deal of commonalities with conceptual modelling in general. We draw a parallel between the concepts proposed here for context and those defined elsewhere for conceptual models based on foundational ontologies [51, 72, 82]. A foundational ontology provides a rich set of basic concepts for representing conceptualizations that are truthful to reality. This means that

conceptual models obtained using well-founded concepts aim at characterizing as precisely as possible the domain they represent.

A foundational ontology, often called an *upper level ontology*, defines a range of top-level domain-independent ontological categories, which form a general foundation for more elaborated domain-specific ontologies [52]. We use the concepts proposed in [51] as foundations to categorize our concepts for conceptual modelling of context. In the sequel we present these concepts.

### 5.2.1 Universals and individuals

*Universals* and *individuals* are fundamental categories that have been considered in our modelling abstractions. Individuals are elements that exist in reality, possessing a unique identity. Universals, on the contrary, are space-time independent patterns of features, which can be realized in a number of different individuals. Every individual instantiates at least one universal. Intuitively, individuals refer to instances, while universals refer to types.

Individuals are classified in *endurants* and *perdurants*. Endurants are individuals that endure over time, i.e. they *are in time*. Examples are a person, a house, a room, a table, and so forth. Perdurants, on the contrary, are individuals composed by temporal parts, i.e. they *happen in time*. Examples of perdurants are a conversation, a football game, and a business process. We focus in this thesis on an *ontology of endurants*. Endurants are further classified into *substantials* and *moments*, which are discussed in the following section.

*Figure 5-1* depicts a fragment of the foundational ontology. A *thing* represents any conceivable and perceivable element in the world. Things can be categorized as *sets* or a *urelements.* Urelements are things without any set-theoretical structure in their build-up. Constructs from the set-theory, such as the membership relation ($\in$) and the inclusion relation ($\subseteq$) cannot be applied to urelements. Examples of urelements are a person, or the Atlantic Ocean. A set, on the contrary, is a thing that has other things as members (in the sense of set theory).

Urelements are classified into individuals and universals, which have been previously discussed.

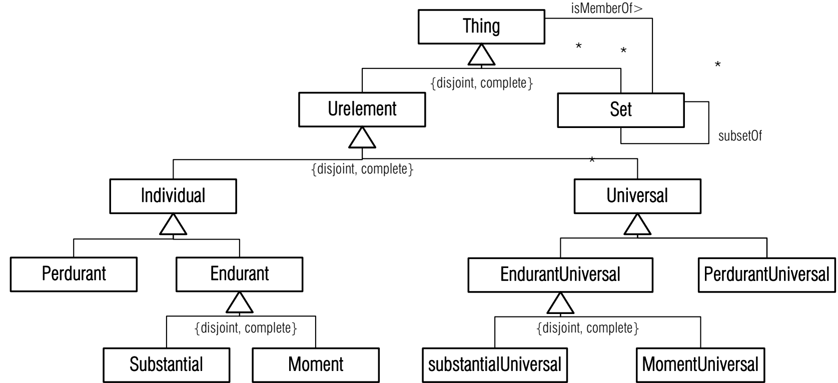



*Figure 5-1* Fragment of foundation ontology

### 5.2.2 Substantials and moments

Endurants can be categorized as *substantials* or *moments* [82]. Substantials are individuals that exist by themselves; this implies that a substantial is existentially independent from other individuals. Existential independence means that "an individual A is existentially independent from an individual B if and only if it is logically possible for A to exist even if B does not exists" [51]. A substantial is an endurant that does not inhere in another endurant, i.e. which is not a moment. Examples of substantials include a person, a dog, a house, the Atlantic Ocean, and so forth.

A moment is an individual that *existentially depends* on other individuals to exist, named its *bearers*. In addition, a moment should also *inhere* on its bearer(s), the way mood inheres in a person and a smile on a face. The inherence relation, sometimes called *ontic* predication, connects moments to the substantials which are their bearers. For example, it connects a smile to the respective face, or the charge in a specific conductor to the conductor itself.

As depicted in *Figure 5-2*, moments can be classified into two categories, namely intrinsic and relational moment.

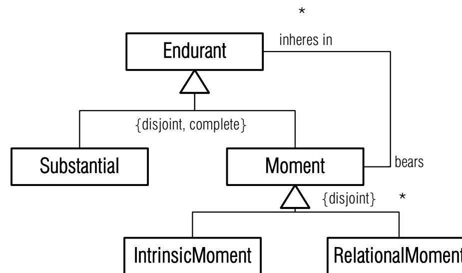



*Figure 5-2* Intrinsic and relational moments

### 5.2.3 Intrinsic moments

An intrinsic moment is an individual that inheres in a single entity. In this thesis we are interested in a particular kind of intrinsic moments called *quality.* Quality is an intrinsic moment that can be mapped to a value (*quale*) in a quality dimension. A quality dimension defines the possible set of values a quality type can be associated with.

The theory presented in [51] promotes the idea that "for each quality type there is an associated quality dimension in human cognition. For example, *height* and *mass* are associated with one-dimensional structures with a zero point isomorphic to the half-line of nonnegative numbers. Other qualities, such as color and taste, are represented by several dimensions. For instance, taste can be represented as tetrahedron space comprising the dimensions of saline, sweet, bitter and sour."

The term quality structure is used to refer to both quality dimensions and quality domains. In addition, the term *quality universal* refers to those moment universals that are associated with a quality structure, and a *quality* is an individual that instantiates a quality universal.

*Figure 5-3* depicts an example of a substantial, one of its qualities, and the corresponding quale. This example shows an apple *a* (substantial), its weight *w* (quality), and *q,* which is the value of the weight in a quality dimension, for example, a kilogram dimension. The relation *i* represents the existentially dependence relation from the weight *w* towards the apple, and the relation *ql* represents the mapping of the quality onto a specific quality dimension.

*Figure 5-3* Substantial, qualities and qualia

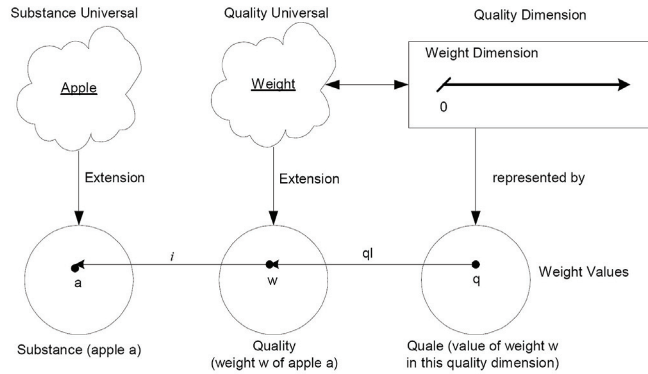
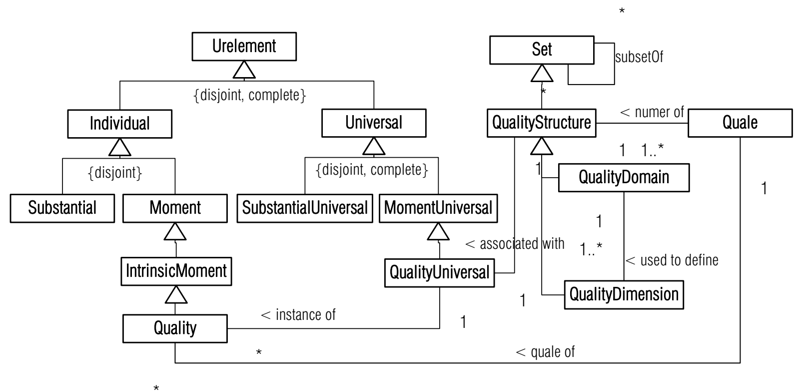



*Figure 5-4* depicts a fragment of the foundational ontology focusing on the concepts introduced in this section.

*Figure 5-4* Fragment of foundational ontology focusing on quality, quality structures and qualia

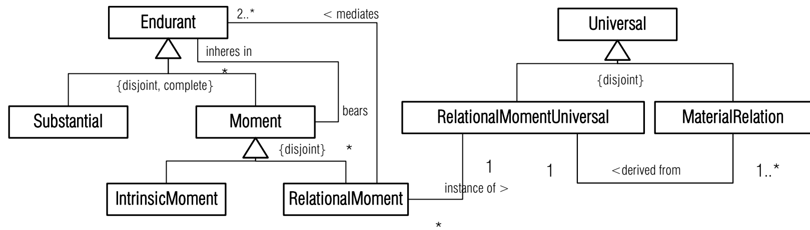



## 5.2.4 Relational moments

A *relational moment* is an individual that connects a plurality of other individuals. Examples of relational moments are a flight connection that connects airports, or a marriage contract that connects two persons. The individuals connected by relational moment are called *relata*, and the relation held between them is called a *material relation*. For example, the relation between the persons that are connected through a marriage contract is a material relation.

*Figure 5-5* depicts a fragment of the foundational ontology focusing on relational moment. A relational moment universal is a universal that only has relational moment individuals as instances.

*Figure 5-5* Fragment of foundational ontology focusing on relational moment




### 5.2.5 Formal relations

Material relations are not the only means by which one can establish the relation between individuals. Conceptual modelling theories also define the notion of *formal relation*. Formal relations hold between two individuals directly, without any intervenient individual. Examples of formal relations are: *greater than*, *taller than, older than* and *subset of*. The immediate relata of such relations are qualities [82], i.e. formal relations are defined in terms of their *relata qualities*.

### 5.2.6 Situations

Modelling the application's universe of discourse allows application designers to represent all possible state-of-affairs in the application's universe of discourse without discriminating particular situations that may be of interest to applications. For example, while we may capture in a conceptual model that a person may be married to another person, it is not the objective of the conceptual model to make statements about particular instances of persons. Therefore, we do not explicitly represent in a conceptual model that John is married to Alice, or that John has been married to Alice for 10 years. In order to enable the representation of particular state-of-affairs, we introduce the concept of *situation*. Situations are composite concepts whose constituents are the elements of our ontology, i.e. substantials and moments. Situations are genuine ontological elements that are composed by other elements. Examples of theories that define situations are [26, 55].

We have extended the foundational concepts presented in [51] in order to support situations. *Figure 5-6* extends *Figure 5-5* with the concept of situation: situations are individuals, which are composed by other individuals. Similarly, situation universals are composed by other universals.

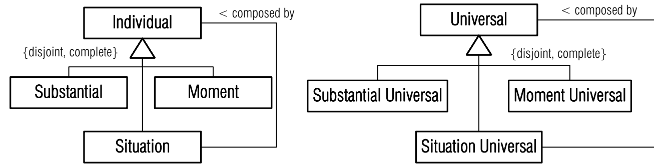



*Figure 5-6 Fragment of f*oundational ontologies including situations

## 5.3 Foundational Context Concepts

### 5.3.1 Entity and Context

Considering the foundational ontologies described in the previous sections and our definition of context, we argue that *entity* and *context* types should be classified into *substantial universal* and *moment universal*, respectively. Since entities do not inhere in other entities, they should be classified as substantials. On the contrary, contexts always inhere in other entities, and therefore, they should be classified as moments.

We define a universal for entities and a universal for context, namely, *Entity* and *Context*, respectively. For example, the entity type *Person* and the context type *Location* are universals, while John and his actual location are individuals (instances of these universals), respectively. We focus here on context models that capture the general aspects of context, and therefore, we only represent universals in our figures.

*Figure 5-7* extends the foundational concepts presented earlier with our *foundational context concepts*. The class Entity (representing the concept of entity) extends the SubstatialUniversal class, while the class Context (representing a particular context condition) extends the MomentUniversal class. The relationship between classes context and entity is characterized by the association ends hasContext and isContextOf, which are subsets of association ends "bears" and "inheres in" described between moments and endurants (*Figure 5-5*). We focus in our approach on foundational context concepts that capture the general aspects of context, and therefore, we only represent universals. For the sake of simplicity, we suppress the term "universal" from our models.

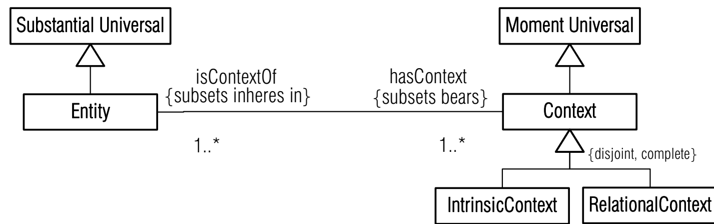



*Figure 5-7* Fragment of the foundational context concepts

The UML constraint {subsets <end>} indicates that an association end constrains the possible values of association end <end>. For example, the possible values of association end hasContext are a subset of the possible values of association end bears, which is defined between classes Endurant and Moment. In ontology terms, a subsets association is a *subproperty*.

We distinguish two categories of context, namely intrinsic context (IntrinsicContext) and relational context (RelationalContext).

Intrinsic context defines a type of context 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 building.

Relational context defines a type of context that depends on the relation between distinct entities. An example of relational context is Containment, which defines a containment relationship between entities, such as an entity building that *contains* a number of entity persons.

This categorization of context is analogous to the ontological categories of moment defined in the previous sections, which classify moments into intrinsic or relational. Similar to our definition, an intrinsic moment inheres in a single individual, while a relational moment inheres in a plurality of individuals.

### 5.3.2 Context situations

As part of our foundational context concepts, we have also included the concept of *context situations*, which are elements composed of contexts and entities. Context situation universals (or types) aim at characterizing situations with similar properties. For example, the context situation universal "John is within 50 meters from Alice" consists of all situation individuals in which the distance between John's and Alice's location values is smaller than 50 meters. Similarly, the context situation universal "Person is within 50 meters from another person" consists of all situation individuals in which the distance between any two persons' location values is smaller than 50 meters. By defining context situation universals, the application designer is able to generalize situations of interest, as opposed to cumbersomely specifying situations that are only applicable to a limited set of entities' instances.

As depicted in *Figure 5-8*, a context situation specialises a situation universal and is composed of *entities* and *contexts*.

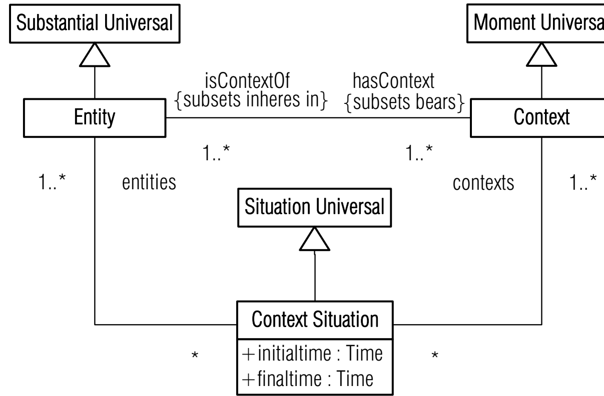



*Figure 5-8* Fragment of the foundational context concepts including context situations

A context situation exhibits temporal properties, such as the time interval during which the situation holds. This aspect is in line with the ontological definition of situations discussed in [55], which defines a situation as a snapshot view of some part of the world. In this theory, a situation is *framed* by a *chronoid*. Chronoids are ontological entities that define a temporal duration. As an example, consider the situations "John is married to Alice" and "John and Alice are divorced". From time $t^0$ to $t^{10}$ (e.g., for 10 years) John has been married to Alice. During this interval, at any time (a snapshot), the situation "John is married to Alice" holds. We can say that the situation "John is married to Alice" is framed by a chronoid that refers to the time interval [$t^0$, $t^{10}$]. Suppose the situation "John and Alice are divorced" is framed by another chronoid defined by the interval [$t^i$, $t^f$]. Since the marriage situation is a pre-requisite for the divorce situation, and a couple cannot be married and divorced at the same time, we can explicitly define that $t^i > t^{10}$.

A context situation chronoid is defined in our context modelling approach by means of initial and final times, represented by the attributes initialtime and finaltime (*Figure 5-8*). The initialtime attribute captures the moment a situation begins to hold, and the finaltime attribute, the moment a situation ceases to hold. Since we capture the finaltime, our model represents past occurrences of situations. We also include temporal operations for relating situations in their occurrence intervals, such as precedence, overlapping, and post-occurrence. These operations are defined in OCL 2.0 [90] in terms of initial and final times, and can be used in the definition of situations. Section 5.5 further discusses temporal aspects.

We have identified a range of situation type patterns that are relevant for context-aware applications. These patterns involve the different kinds of context (intrinsic and relational), entities, and formal relations, which are

the building blocks for composing context situations. We have identified five patterns of context situation types according to the types of context they are composed of. These patterns of situation types are *intrinsic situation*, *relational situation*, *formal relational situation*, *situation of situations* and *combined situations*. *Figure 5-9* depicts these types of situations as extensions of our foundational context concepts. These patterns are described in the sequel.

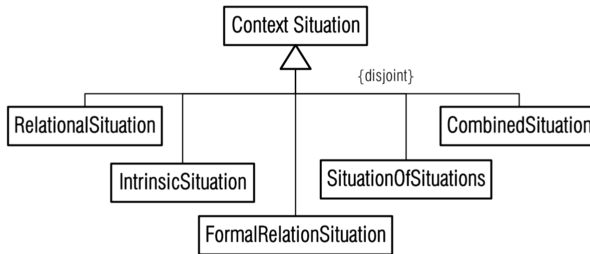



*Figure 5-9* Fragment of the foundational context modelling concepts including context situation patterns

### *Intrinsic situation*

Intrinsic situation defines a type of situation that involves only intrinsic context types. An intrinsic situation is composed by a unique entity and part of its intrinsic contexts. An example of an intrinsic situation is the situation *fever*, which can be defined as a person (of type entity) who has his/her temperature (of type context) above 38 degrees Celsius.

*Figure 5-10* depicts the IntrinsicSituation type as an extension of our foundational concepts. The association ends intContexts and intEntity specialize the association ends contexts and entities defined between the superclasses ContextSituation and Context, and ContextSituation and Entity, respectively. We use the UML *subsets* constraint to define association end specializations.

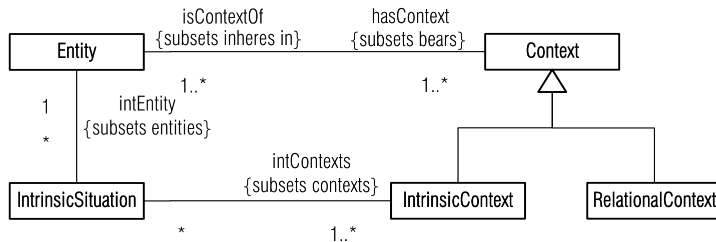



*Figure 5-10* Intrinsic situation

### *Relational situation*

Relational situation defines a type of situation that involves only relational context types. A relational situation consists of at least two entities and part of their relational contexts. An example of relational situation is the situation *friends*, which defines a situation in which two persons (of type entity) participate in a friendship (of type relational context).

*Figure 5-11* depicts the RelationalSituation type as an extension of our foundational concepts. Similarly to the intrinsic situation models, the

associations regarding the RelationalSituation class specialize the associations defined for its superclass ContextSituation, which are presented in *Figure 5-8*.

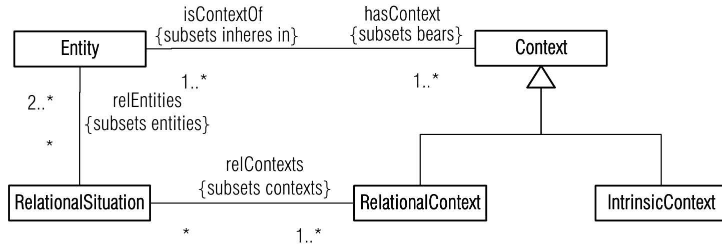



*Figure 5-11* Relational situation

### *Formal relation situation*

Formal relation situation defines a type of situation that consists of at least two entities and at least two or more qualities (intrinsic contexts) such that these qualities are comparable. Two qualities are comparable if they can be associated to a common quality dimension. An example of formal relation situation is the situation *containment*, which defines a situation in which a person (of type entity) is physically contained in a building (of type entity). In order to verify the containment relation, quality values of both entities (for example, their geographical location), should be compared. *Figure 5-12* depicts the FormalRelationSituation type as an extension to our foundational concepts.

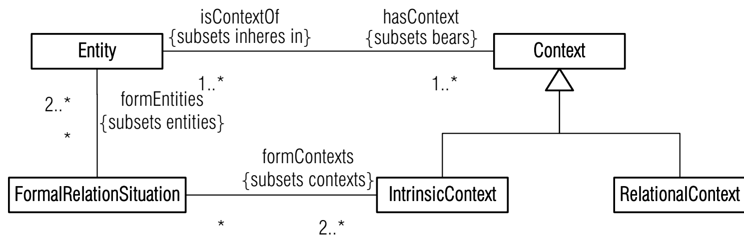
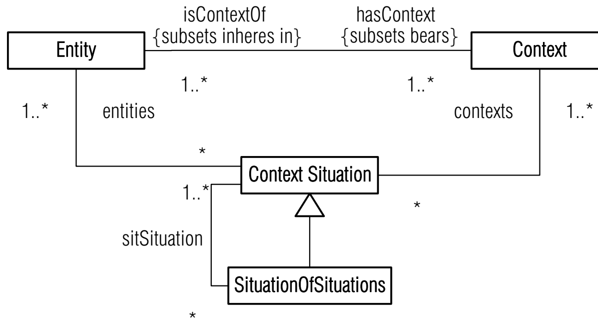



*Figure 5-12* Formal relation situation

### *Situation of situations*

Situation themselves may be composed of other situations. A Situation of situations defines a type of situation that is composed of other situation types. An example of situation of situations is the situation *recurrent fever*, which specifies that a person (of type entity) has had fever (of type situation fever) in the past 5 days. *Figure 5-13* depicts the SituationOfSituations type as an extension of our foundational concepts.

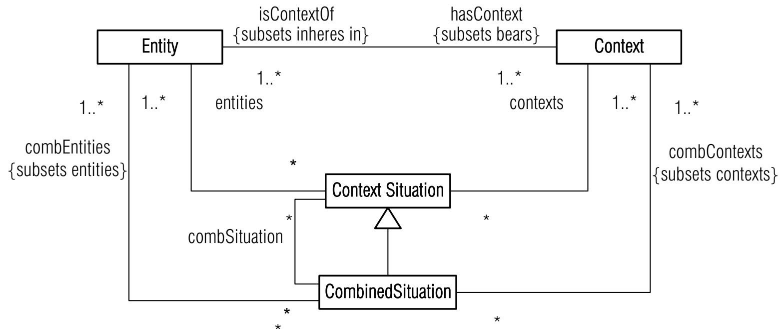
*Figure 5-13* Situation of situations




***Combined situation***

Combined situations are the situations that combine the context types and the situations types we have defined so far. While in the definition of a situation of situations a situation type is always involved, in a combined situation definition such characteristic does not apply. Situations and context types can be freely composed, such that particular application's state-of-affairs are represented. Take as an example a particular situation in which a person is located in a shopping mall and is purchasing a product while talking to some friends. One can define this situation with combined situations by composing entity types (persons and shopping mall), situation types (containment and friendship situations) and intrinsic context types (person talking). *Figure 5-14* depicts the CombinedSituation type as an extension of our foundational concepts.

*Figure 5-14* Combined situations




### 5.3.3 Context foundations

In the previous sections we have defined the foundational concepts that can be extended by context-aware application developers with application- (or domain-) specific concepts. These concepts should be used as a starting point for context-aware application design. In the following sections we provide guidelines on how these foundational concepts can be extended

with specific concepts. *Table 5-1* summarizes the concepts we have discussed so far.

*Table 5-1* Foundational context concepts

| Foundational context concepts | Description |
|---|---|
| *Entity* | an object that beares context |
| *Context* | a particular condition that inheres in an entity |
| *Intrinsic Context* | a particular type of context that belongs to the essential nature of a single entity |
| *Relational Context* | a particular type of context that depends on the relation between distinct entities |
| *Contextual Formal Relation* | a relation that holds directly between two or more entities' intrinsic values (qualities) |
| *Context Situation* | a composite concept that defines particular application's state-of-affairs. It can be composed of entities, contexts, and other situations |
| *Intrinsic Situation* | a context situation composed of a single entity and one of its intrinsic contexts |
| *Relational Situation* | a context situation composed of at least two entities and their pertinent relational contexts |
| *Formal Relation Situation* | a context situation composed of a single entity type and two or more of its intrinsic contexts |
| *Situation of Situations* | a context situation composed of other context situations |
| *Combined Situations* | a context situation composed of other contexts, entities, and other context situations |

## 5.4 Context Models

So far we have discussed the foundational context concepts that can be beneficially used in the design of context-aware applications. We have not yet discussed concepts that are specific to particular applications or family of applications, which are called here application *domains*. In this section we discuss context models that represent the context conditions that are relevant for particular context-aware application's (or application domain's) universe of discourse, and therefore, are application-specific or domain-specific. The examples presented below show how to specialize the foundational context concepts with specific concepts. In addition, these models form the basis for defining *context situations*, which are introduced in section 5.5.

A domain-specific model defines concepts that are shared among the applications that belong to that domain. Some of the domain-specific models may be specialized by particular applications to accommodate the concepts that are application-specific and not shared with other applications in that domain. In this section we show examples of both

application and domain models without explicitly distinguishing between them.

So far we have defined the concepts for context and situation modelling without defining a language with which application designers can specify application specific context and situation models. We specialize UML class diagrams for that purpose introducing a UML *profile* that captures the concepts which are specific to our models, i.e. the concepts summarized in *Table 5-1*. A UML profile is a so called lightweight extension of the UML metamodel and consists of stereotypes. Stereotypes allow the extension of UML permitting the creation of new model elements, derived from existing ones [91]. We have created stereotypes for each element of our foundational context concepts (*Table 5-1*), which form our UML profile. These elements appear as primitive constructs for context-aware application designers to build specific context models. For example, when an application designer defines the concept Person as a UML class, it is stereotyped as «Entity», which indicates that this concept is of type Entity according to our foundational context concepts. Similarly, if the location concept is defined as a UML class, it is stereotyped as «IntrinsicContext», which indicates that this concept is of type intrinsic context according to our foundational context concepts.

### 5.4.1 Entity types

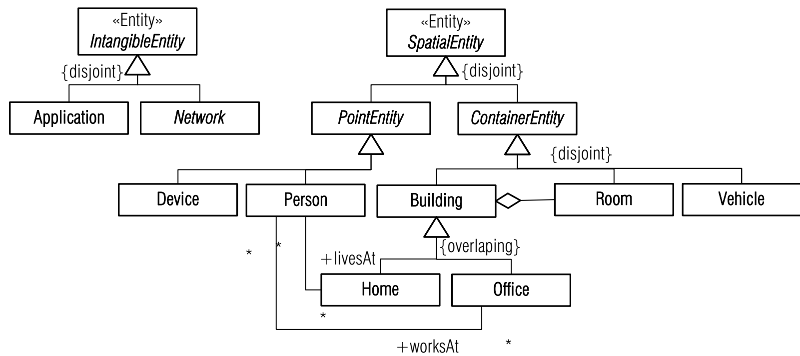
*Figure 5-15* depicts some examples of domain specific (or application specific) entity types such as SpatialEntity and IntangibleEntity. We use the stereotypes to denote the classification of a domain (or application) specific concept with respect to the foundational concepts. For example, by using the stereotype «Entity» for a class, the designer explicitly specifies that this class inherits the characteristics of the foundational concept Entity (depicted in *Figure 5-7*).

Spatial entities represent tangible objects, such as a person, a device, a room or a building. Intangible entities represent intangible objects such as an application and a network. A particular type of spatial entity is the ContainerEntity, which is capable of physically containing other entities. Container entities have different shapes and dimensions, such as the height, width and depth of a rectangular parallelepiped, or the radius of a sphere. In addition, container entities have a *centre of mass*, which defines a point in the centre of the container object. We regard the geographical location of a container entity, the geographical location of the container's centre of mass.

The counterparts of container entities are the PointEntity types, which have no volume dimensions, behaving like a point in the space. In *Figure 5-15* we show person and device as examples of point entity types. Although these entities in reality have volume, it is the application developer who decides

whether the entity type is relevant as a container or as a point entity. Therefore, depending on the application's universe of discourse, entity types may be represented differently.

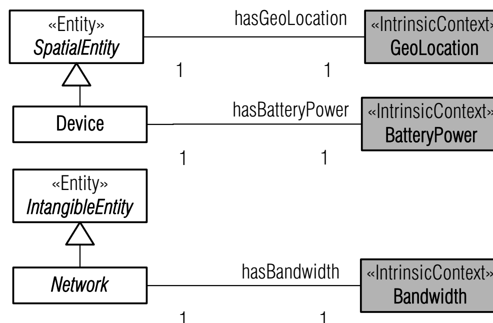
*Figure 5-15* Examples of entities




### 5.4.2 Intrinsic context types

Intrinsic context defines a type of context that inheres in a single entity, which is the *bearer* of the intrinsic context. *Figure 5-16* and *Figure 5-17* depict examples of intrinsic context types. Geographic location (GeoLocation) is context that inheres in all spatial entities. Spatial entities are *bearers* of GeoLocation. Similarly, battery power (BatteryPower) inheres in a device. Analogous reasoning can be applied to other context types depicted in *Figure 5-16* and *Figure 5-17*.

*Figure 5-16* Intrinsic context types




In order to represent a concept as an entity types or as an intrinsic context type, the stereotypes «Entity» and «IntrinsicContext» are used, respectively. The association ends between entity and context types are regarded as the association ends hasContext and isContextOf, i.e. in *Figure 5-16*, the association end hasGeoLocation is stereotyped as «hasContext». We omit stereotypes on association ends for the sake of clarity of the models.

All intrinsic context types discussed in this thesis are classified as the ontological notion of *quality universal*. The geographical location of an entity

is an example of quality, whose quality dimension is defined by all possible values in a geographical coordinate system.

The quality of an entity is an intrinsic objectified property of that entity, thus, even if two entities are co-located, they do not have the same location quality in the strong sense. Co-location depends on the granularity of associated quality dimension. For instance, take two different quality dimensions *Q*, *Q'* associated with the quality universal location such that *Q* = {*list of names of civil locations*}, *Q'* = {*precise GPS location value space*}. Under these circumstances, we can have that two entities are considered co-located in the quality space *Q* but not in *Q'*. In other words, the accuracy of our comparisons of entities' intrinsic properties depends on the precision of our quality dimensions. Quality dimensions are represented as datatypes in our models.

*Figure 5-17* presents examples of intrinsic context types of a person, such as the person's current activity, mood and mental state. In fact, these context types are quite subjective and difficult to measure. However, one could conceptualize an objective notion for these context types in a context-aware application, by enumerating the possible values (quality dimension) with which each of these types may possibly be associated. For example, we may say that the possible values of a person's mood are: *happy*, *sad*, *bored*, *tired* and *moody*; and the possible values of a person's current activity are: *working*, *dancing* or *attending a meeting*.

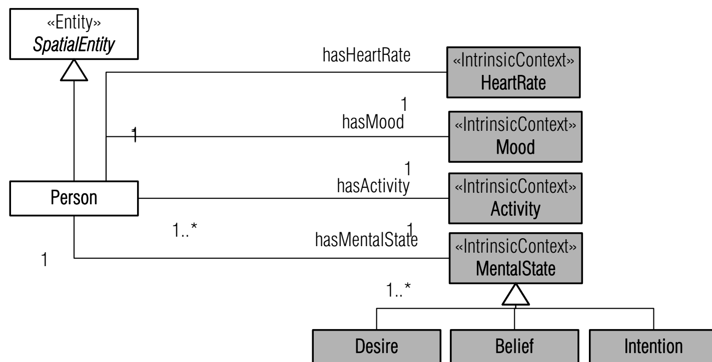



*Figure 5-17* Intrinsic context types for persons

*Figure 5-18* shows how environmental characteristics can be modelled by using intrinsic context types associated with a ContainerContext. Examples of container context types are noise level and temperature of a room and humidity of a car. These context types are also *qualities*, and therefore, quality dimensions should be specified for each of them. The quality dimension of relative humidity, for example, comprises the values between 0 and 100 (percentage values). Quality dimensions are represented as UML *datatypes* in our diagrams, as depicted in *Figure 5-18*. Datatypes are

appropriate to represent quality dimension since instances of these types are values in a value space.

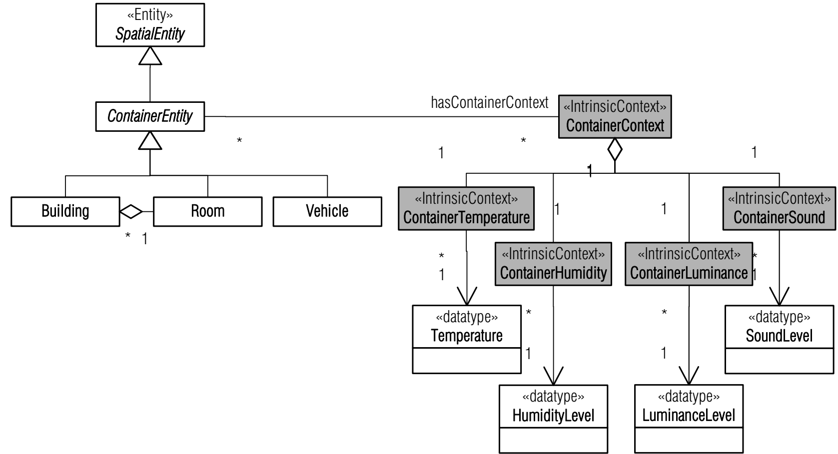



*Figure 5-18* Intrinsic context types for container entities

In some scenarios, depending on the modelling choices, context information may be classified as either intrinsic or relational. Take for example the entity's civil location (country, state, city, street and house). As depicted in *Figure 5-19*, civil location can be classified as intrinsic context, in which the value of a civil location is mapped to a quality dimension defined as the CivilLocaiton datatype.

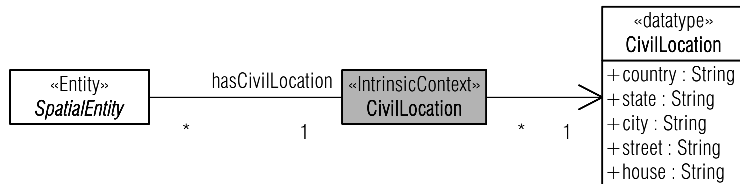



*Figure 5-19* Civil location as intrinsic context

However, it may be necessary to treat country, state, city, street and house as entities themselves, since one may be interested in properties of these entities, such as the number of persons in a house, the holidays of a country and the traffic intensity on a street. In such scenarios, a civil location depends on the existence of a set of entities, and, therefore, should be classified as relational context (see section 1.2.2), as depicted in *Figure 5-20*.

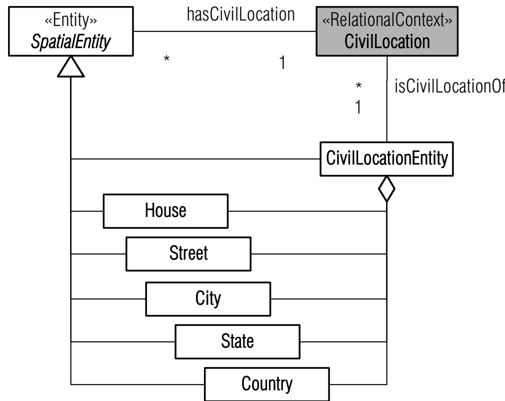



*Figure 5-20* Civil location as relational context

### 5.4.3 Relational context types

While intrinsic context information inheres in a single entity, relational context information inheres in a plurality of entities. In order to specify that an application-specific concept is a relational context, the stereotype «RelationalContext» is used. The relation that holds between bearers of a relational context is a *material relation*. For example, the relation that holds between devices and channels through ChannelAvailability (see *Figure 5-21*) is a material relation. A material relation is defined directly between the entities participating in a relational context. We have simplified the models in the remaining examples of the thesis, such that a material relation is not explicitly shown. *Figure 5-21* and *Figure 5-22* depict examples of relational context types.

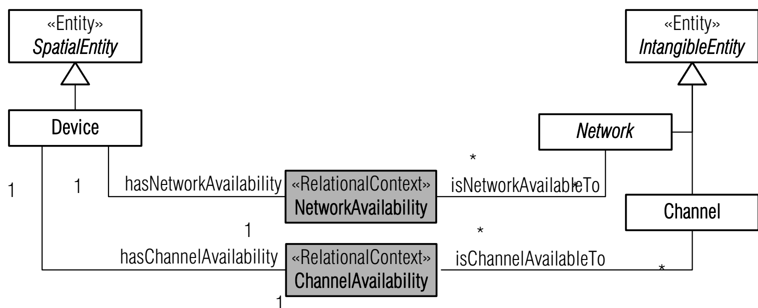



*Figure 5-21* Channel and network availability relational context types

Relational context may be used to relate an entity to the collection of entities that play a role in the entity's context. Examples of relational context types are DeviceAvailability, NetworkAvailability, SocialNetwork and ChannelAvailability. The DeviceAvailability relational context relates a person to a collection of devices that are available to that person. NetworkAvailability relates a device to a collection of networks that are available through that device, SocialNetwork relates a person to the collection of persons interacting with

that person by any communication channels, and ChannelAvailability relates a device to a collection of communication channels supported by that device (e.g., e-mail, voice and SMS).

*Figure 5-22* Containment, device availability and social network relational context types

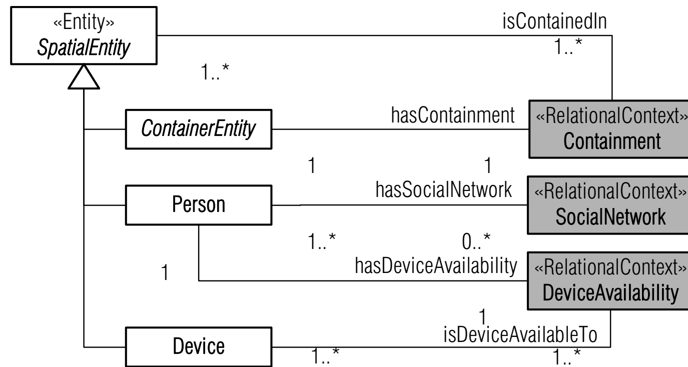



*Figure 5-22* depicts another example of a relational context, the Containment context, which represents a direct containment relationship among spatial entities. More specifically, a ContainerEntity such as a building, a room or a vehicle may be associated with a containment relational context, which may in turn contain a set of spatial entities. A containment chain is created with the condition that every contained entity physically fits in its respective container entity.

Intuitively, relational context allows us to navigate the context model from an entity to the contexts of entities that are related to this entity through the relational context, still maintaining the separation of the concerns between entity and context. Consider the following example involving the entity types Person, Device and Channel. Suppose that John (of type Person) is related to his PDA and phone (of type `Device`) through DeviceAvailability. John's PDA is related to e-mail (of type Channel) though ChannelAvailability, and John's phone is related to a voice channel also through ChannelAvailability. Therefore, we can conclude that John is (indirectly) related to certain e-mail/voice channels. This approach can be beneficially used in reasoning activities.

The participants of a relational context might be of the same type, for example, the participants of a SocialNetwork relational context are of type Person. In some scenarios this participation is symmetric, i.e. each entity participates in the relational context in the same way. For example, in *Figure 5-22*, the persons participating in the SocialNetwork relational context (e.g., a *friendship*) equally contribute to the social network. However, there are scenarios in which it is necessary to distinguish the participation of entities in a relational context. As can be seen in *Figure 5-22*, the participation is distinguished by means of association names. For example, a person participates in the DeviceAvailability relational context by means of a

hasDeviceAvailability association, while device participates in the DeviceAvailability relational context by means of a isDeviceAvailableTo association.

*Figure 5-23* depicts an example of relational context type in which the participants are of the same type, but contributing distinctively to the relational context. The persons in a telephone call participate either as a caller or a callee, which are defined as distinct associations in the relational context.

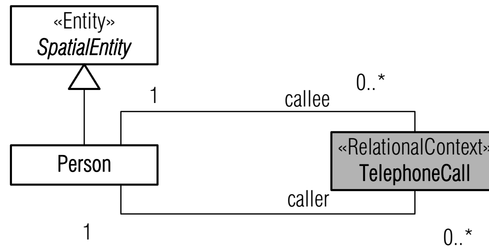



*Figure 5-23* Telephone call relational context

Relational context types may be also specialized in order to meet specific modelling requirements. For example, if we would like to model specific types of social networks, such as a friendship or a social gathering, such as a book circle network. *Figure 5-24* depicts these relational context types.

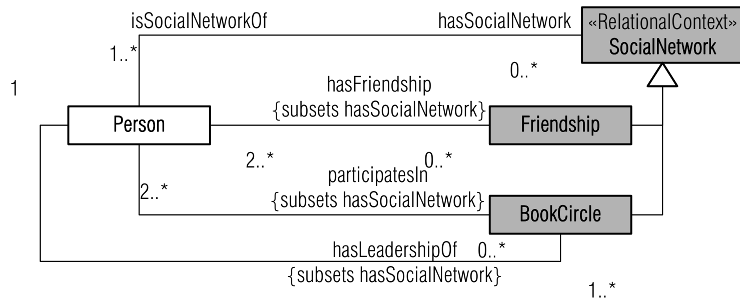



*Figure 5-24* Friendship and book circle relational contexts

Persons equally participate in a friendship; therefore, only association hasFriendship between classes Person and Friendship is defined. In a book circle, on the contrary, a person may participate in different ways, namely as a member (participatesIn association end) or as a leader (hasLeadershipOf association end) of the circle.

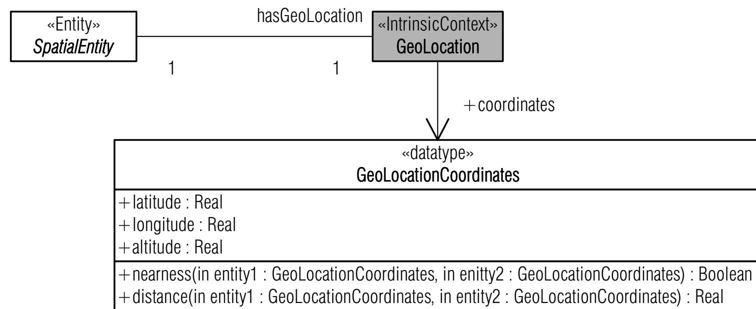
### 5.4.4 Contextual formal relations types

Analogous to the definition of formal relations discussed in section 5.2, contextual formal relations hold directly between intrinsic context types. As opposed to material relations, there is no intervenient individual between the intrinsic context types related by a contextual formal relation. Formal

relations are defined in our models as operations in the data types which represent the respective quality dimension.

*Figure 5-25* depicts an example of two formal relations, namely *nearness* and *distance*. Nearness is a formal relation in which the truth value of an expression such as "John is near Maria" ("nearness" being defined, for example, as within 1 km range) only depends on the values of John's and Maria's locations, which are qualities (intrinsic context). Distance is a formal relation that calculates the distance between spatial entities. For example, the distance (Distance(x,y,z)) can be thought of as a logical construction from the intrinsic context a = location(x), b = location(y), such that z = |valueof(a)-valueof(b)| (Euclidian distance between a and b). Nearness and distance are defined in our models as operations in the GeoLocationCoordinates data type (*Figure 5-25*).

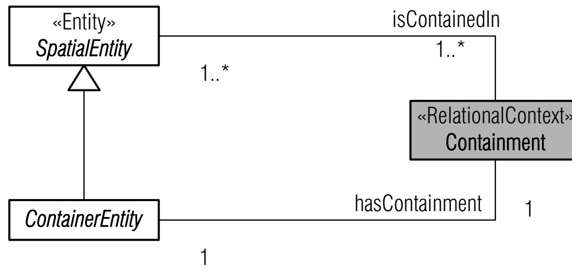
*Figure 5-25* Formal relations




The distinction between material and formal relations are useful in our context models. On one hand, it is possible to derive or infer the truth value of a formal relation solely from the intrinsic context of the related entities. On the other hand, direct inference from intrinsic contexts of the related entities is not sufficient to determine whether a material relation holds.

Relational context and formal relations may be interchanged, depending on the context model adopted. For example, one could adopt a model where the containment relational context is defined in terms of the spatial dimensions of a container and the location of a contained entity, being therefore a formal relation. A different approach is to adopt a model where the containment relational context exists on its own (for example, in a badge system). In such scenarios, there is no need to explicitly conceptualize neither the spatial dimensions of a container nor the location of a contained entity. In this scope, containment is categorized as relational context (and hence, defines a material relation between the container and contained entities). *Figure 5-26* depicts containment as a relational context, and *Figure 5-27* depicts containment as a formal relation.

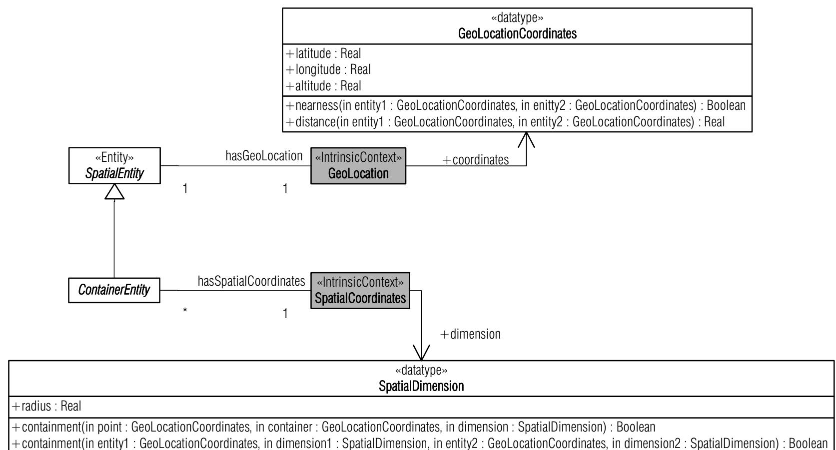
*Figure 5-26* Containment as relational context




In *Figure 5-27*, we define the quality dimension SpatialDimension which describes the dimensions needed to calculate the volume of a container entity. The value of a SpatialCoordinates intrinsic context must be mapped to a value in the SpatialDimension quality dimension. We exemplify the spatial dimension with a spherical shape. Radius and centre of mass are enough to calculate the volume and the position in space. However, other shapes, such as parallelepiped and elliptical shapes could also be defined.

We have defined two containment operations (formal relations): to verify whether the container entity contains another container entity and to verify whether the container entity contains a point entity.

*Figure 5-27* Containment as formal relation




## 5.5 Context Situation Models

We specify context situations using standard UML 2.0 [91] class diagrams which are enriched with OCL 2.0 [90] constraints to define the conditions under which context situations of a certain type exist. For example, *Figure 5-28* depicts the representation of a simple situation type in which a person has fever (his/her temperature is above 38 degrees Celsius).

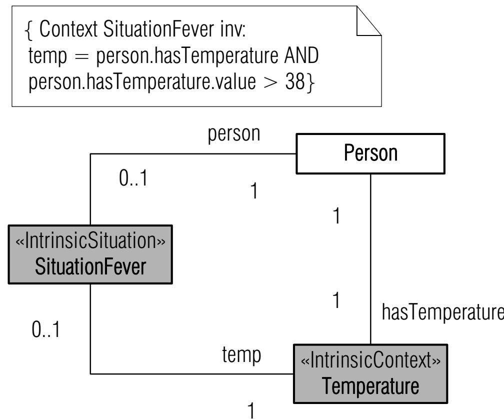



*Figure 5-28* Situation fever

The situation type SituationFever is composed of entity type Person and his/her Temperature, which is an intrinsic context type. The OCL invariant defines a predicate that must hold for all instances of SituationFever. The cardinality also constrains situation instances, for example, instances of SituationFever should be associated with a person, and a temperature. The OCL invariant further constrains instances of SituationFever by defining that the temperature of the person should be greater than 38 degrees Celsius. The "context" keyword in the OCL invariant is a reserved OCL primitive that defines the class for which the constraint should be applied.

The association ends between a situation type and an entity type, and between a situation type and a context type, are classified according to the respective association ends defined for that pattern of situation type. For example, since the situation fever type is an intrinsic situation, the association ends person and temp are classified as «intEntity» and «intContexts» (defined in *Figure 5-10*) respectively. We omit association end stereotypes for the sake of clarity of the models.

The invariants as presented in our situation type figures are violated for past occurrences of situations, i.e. these invariants would never allow past occurrences of situations. In order to avoid that, we should include, for each invariant, a disjunction with a predicate that verifies whether this situation is a past occurrence (not finaltime.oclIsUndefined()). We omit this predicate in the rest of the thesis (except when explicitly indicated) for the sake of readability. In the sequel we discuss application specific examples of each of the situation patterns discussed in section 5.3.2.

### 5.5.1 Intrinsic situation types

Intrinsic situation defines a type of situation that involves only intrinsic context types. An example of an intrinsic situation type is "person has fever" (SituationFever), which is depicted in *Figure 5-28*. Another example is

the situation type (SituationAvailable) that captures the availability and willingness to communicate of MSN and Skype users.

*Figure 5-29* depicts a fragment of the structural context model that represents the MsnStatus and SkypeStatus intrinsic context types, which model the user's communication status while using MSN and Skype, respectively. A person, while playing the role of MsnUser, is associated with MsnStatus context type, and while playing the role of SkypeUser, is associated with SkypeStatus context type. The enumeration data types SkypeStatusEnum and MsnStatusEnum define all possible values for SkypeStatus and MsnStatus, respectively.

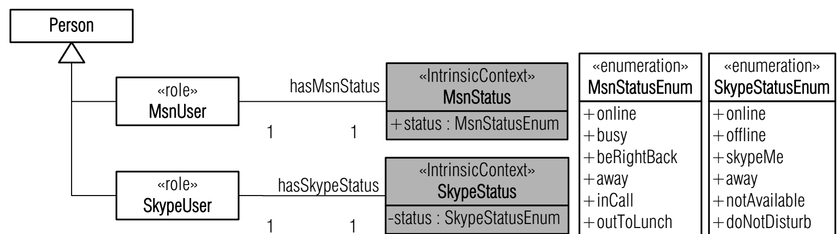



*Figure 5-29* Context model used in the situation available specification

*Figure 5-30* depicts a situation model which builds on the context model presented in *Figure 5-29*, defining the situation type SituationAvailable.

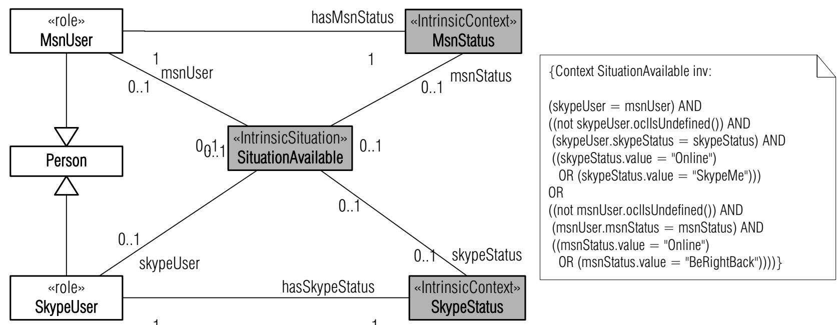



*Figure 5-30* Situation available specification

The OCL invariant in this diagram is a predicate that must hold for all instances of SituationAvailable. It defines that instances of SituationAvailable must be either associated with a user available in Skype (with SkypeStatus set to Online or SkypeMe) or a user available in MSN (with MsnStatus set to Online or BeRightBack). The OCL operation oclIsUndefined() is part of the OCL standard library and tests whether the value of an expression is undefined.

### 5.5.2 Relational situation types

Relational situation defines a type of situation that is composed by relational context types. The following example discusses a situation (SituationConnection) in which a device has established a connection (relational

context type) to each of the two network types, WLAN, and Bluetooth (entity types). By explicitly modelling the connections as relational context, we are able to assign properties to these connections, such as access rights and negotiated QoS.

*Figure 5-31* depicts the structural context models representing the types and relationships that are relevant for this example. According to this diagram, a Device may be connected to a Network through the relational context Connection.

*Figure 5-31* Context model for situation connection

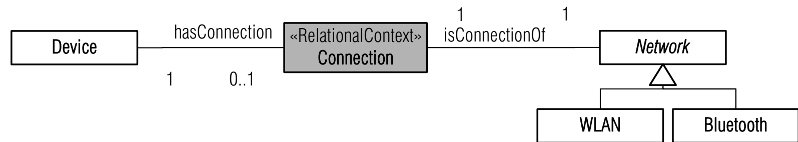
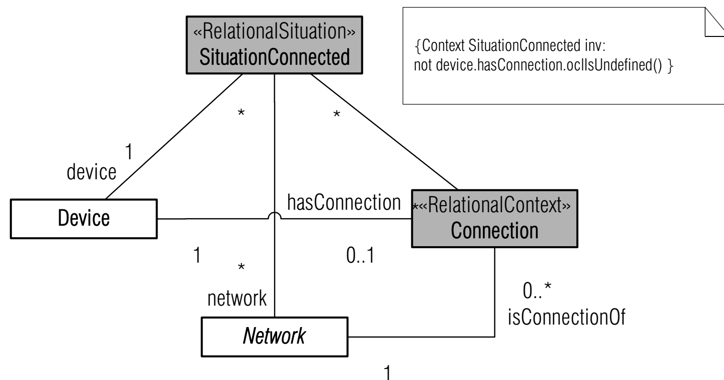



*Figure 5-32* depicts the situation type SituationConnected. The OCL invariant defines that instances of this situation must be associated with at least one connection object.

*Figure 5-32* Situation connection definition




### 5.5.3 Formal relation situation types

Formal relation situation defines a type of situation that consists of at least two entities and at least two or more intrinsic contexts. *Figure 5-33* shows an example of situation involving two entities and their intrinsic contexts. Their locations are compared such that instances of SituationWithinRange hold if two spatial entities are located within a certain range (defined as an attribute of the SituationWithinRange class). This model builds on the context model defined in *Figure 5-25*, which specifies the formal relation distance, as an operation of the GeoLocationCoordinates data type.

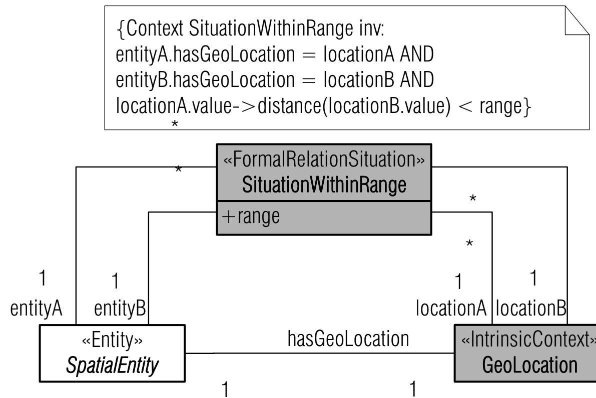



*Figure 5-33* Situation within range specification

### 5.5.4 Situation of situations types

Situation of situation types are composed of other situations types. Suppose we would like to know when a device switches from a WLAN connection to a Bluetooth connection in order to set new quality of service parameters. Since SituationConnected has been already defined in *Figure 5-32*, in order to detect SituationSwitch, we would have to verify whether SituationConnected held in the past for network WLAN, and currently holds for network Bluetooth. We may add the additional constraint for the handover time which should not be longer than one second. This temporal constraint is realized by means of the initial and final times attributes. This example is depicted in *Figure 5-34*, showing that SituationSwitch can be modelled by composing multiple occurrences of SituationConnection, one called wlan, and the other called bluetooth.

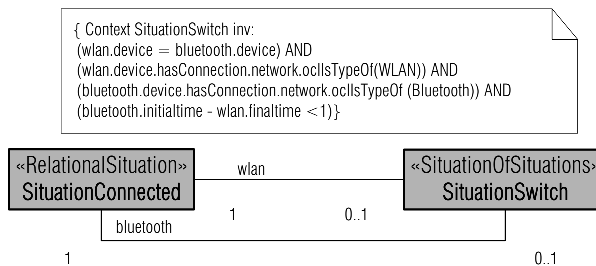



*Figure 5-34* Situation switch specification

We can also define situations that depend on the duration of other situations. Suppose we would like to know when two persons are within certain range for some time, which could mean that they are/were in a meeting. For that, we can use the SituationWithinRange type depicted in *Figure 5-33* with an extra constraint that determines the minimum duration of the situation. The situation type depicted in *Figure 5-35* holds when two persons are/were within a certain range for at least 1 hour (60 minutes).

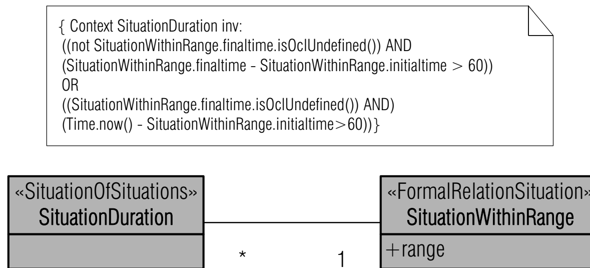



*Figure 5-35* Situation duration specification

We could also slightly change this OCL constraint to allow SituationDuration to hold only when the persons were together in the past, or to hold only when these persons are still together in the present. In order to guarantee that the persons were together in the past, we would keep the first part of the invariant, eliminating the binary logical operator OR. Similarly, to guarantee that the persons are still together in the present, we keep the second part of the invariant, also eliminating the OR operator.

Defining situation of situations is a good practice when specifying the application's state-of-affairs, since it enables modularization of application design. Situation types can be used as building blocks for composing more complex structures of situations. This allows reuse of situation types in the specification of different state-of-affairs.

### 5.5.5 Combined situation types

Combined situations are the situations that combine the context types and the situations types we have defined so far. For example, suppose we would like to define a situation in which a device is capable (with respect to time) of downloading and launching a presentation. In order to define that, we should determine (i) whether the device is connected to a network; (ii) the time necessary to download the presentation given the presentation size and the current network bandwidth; and (iii) whether the device's remaining battery provides enough time to download and give the presentation. *Figure 5-36* depicts an example of such situation, which combines intrinsic context types, relational context types and formal relations.

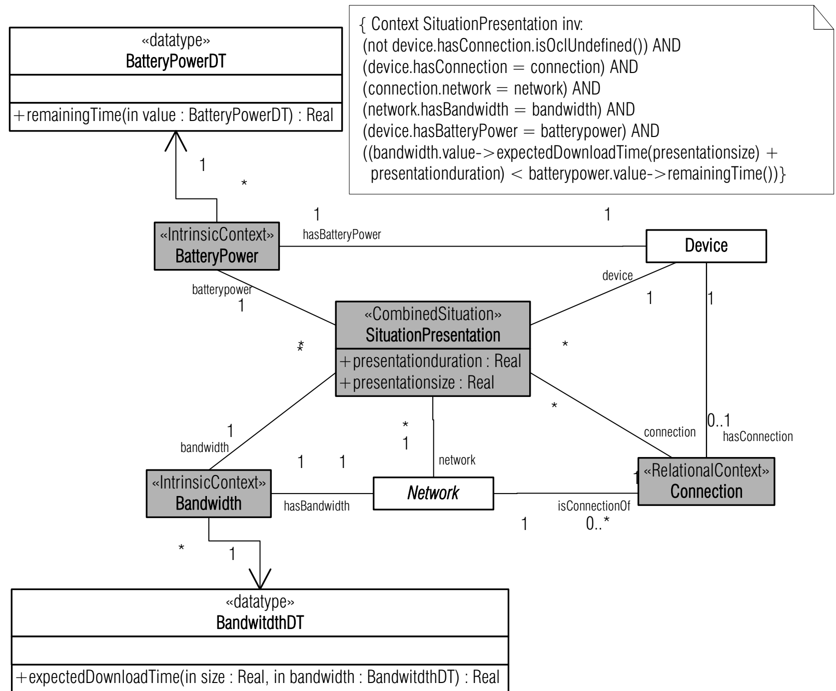



*Figure 5-36* Combined situation specification

The SituationPresentation type has the presentation duration and the presentation size as attributes. These attributes are necessary to know the download time, and the total time necessary to download and give the presentation, respectively. The OCL constraint makes sure that SituationPresentation only holds when the remaining battery time is enough to download and give the presentation. Both operations expectedDownloadTime() and remainingTime() are contextual formal relations.

Although designers are free to use combined situations, they should be aware that complex combined situations are less reusable than simple situation types. It is more efficient to define simple situation types that are more reusable, than defining complex combined situations that less reusable. In addition, it is often the case that complex situation types can be composed of more elementary situation types. In such cases, using situation of situations types is preferred over using combined situation types.

## 5.6 Context Information Models

As discussed in chapter 2, sections 2.1.2, we explicit distinguish between the concepts of context and context information, which leads to two distinct phases in our modelling approach, namely, *conceptual context*

*modelling* and *context information modelling*. So far we have discussed the conceptual modelling phase. The context information modelling phase regards technology-specific aspects in the context models. In addition, we consider the quality of context information (QoC), which is strongly dependent on the mechanisms used to capture the corresponding context conditions from the user's environment (see chapter 2, section 2.2.4).

*Figure 5-37* depicts the relations between the context models and the components that provide context and situation information, namely context sources and managers, respectively. Context information models increment the conceptual models with concepts that vary based on the technology used, such as quality of context. Context sources and managers accept requests and provide responses which are in line with the concepts of both conceptual models and context information models.

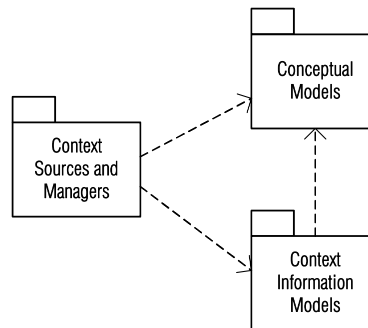



*Figure 5-37* Relations between models and components

Consider an example of a context source that provides location information, which is defined in the scope of the *Context Sources and Managers* package. This context source accepts requests of information types that have been defined in the conceptual models, such as "query GeoLocation of entity John". GeoLocation is intrinsic context defined in our conceptual models. In addition, the location information may present values that qualify this information. Meta-information about quality of context is defined in the context information models. For example, we may want to constraint the query above to only retrieve GeoLocation of entity John, when the probability of correctness of this information is at least 80%. In this case, the concept of *probability of correctness* (meta-information), and how it relates to specific context types are defined in the context information models.

### 5.6.1 Quality of context

Since sensor technology is imperfect by definition, we should provide mechanisms and abstractions that allow us to reason about the quality of the context information captured by sensors. Quality of Context (QoC) is

information that qualifies context and context situations. Specific context and situation types may exhibit particular types of QoC, which are called *quality of context parameters*. Examples of such parameters are *precision*, *probability of correctness*, and *freshness*. These parameters allow us to determine the deviations between the ideal (perfect) information and the information that is actually obtained from the sensors.

Quality of context parameters have been extensively discussed in the literature [9, 48, 111]. Our aim is to use examples of quality parameters from related work in order to demonstrate how they can be incorporated to our modelling approach. We do not intend to extensively discuss quality of context parameters, how they relate to each other, and how they are calculated. We consider the following examples of QoC parameters [111]:

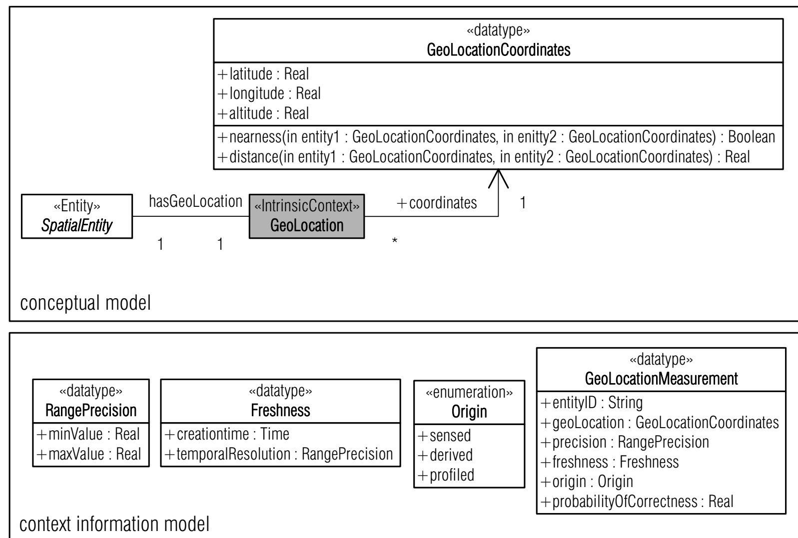
- Precision: the granularity with which context information describes a real world situation. Depending on the type of information, different precision quantification can be used. For numeric values, a range of values can be used to characterize precision. For example, we can say that the temperature of a room is 23 degrees Celsius, with precision of 2 degrees, i.e. the temperature value may vary between 21 and 25 degrees. For temporal types, we may say an event occurred at 15 hrs with precision of 3 hours, meaning that this event may have occurred between 12 and 18 hours;
- Freshness: period of time between the determination of context information, and the time it is delivered to the requester. The freshness of context information can be quantified by creating a timestamp when the information is determined. The requester calculates the freshness based on the current time, and the time when the information has been determined;
- Probability of correctness: the probability that the context information value accurately represents the corresponding real world context. Various factors may contribute to the probability of correctness of context information, such as partial unavailability or malfunction of the sensor technology. Notably, the probability of correctness also varies depending on the precision and freshness parameters. In general, the more precise and fresh the information is, the more accurate is should be.

These quality parameters are highly dependent on the measuring mechanisms used by the context providers (context sources and managers). The context information models aim at identifying the types of providers supported in the application, the context types supported by these providers, and which quality parameters are supported.

In our modelling approach, *measurement datatypes* are created in order to define the serializable (context and situation) information types that are exchanged between components. *Figure 5-38* depicts the

GeoLocationMeasurement datatype, which defines the serializable geographical location information that may be exchanged between components. This datatype defines the attributes (i) entityID, which is a string that uniquely identifies the spatial entity; and (ii) the geoLocation attribute, which are the measured location coordinates of that entity. This datatype also defines the quality parameters applied to a context provider that captures geographical location information, using for example, a GPS device or a GSM triangulation mechanism.

*Figure 5-38* GeoLocation Measurement datatype in the context information model




The RangePrecision datatype defines a quantification mechanism for precision based on range values. The Freshness datatype defines the attributes necessary to quantify the freshness of the information: the timestamp of the context information creation, and the temporal resolution, which defines the precision used for the timestamp value. The Origin datatype specifies how the information has been acquired. This class enumerates the following possible origins for context information, namely *sensed*, *derived* or *profiled*. Context information is *sensed* when acquired directly from sensors, and no reasoning algorithms, such as derivation, extrapolation or aggregation have been applied. Context information is *derived* when some sort of reasoning has been used to derive such information. For example, context information that indicates whether a user is busy or not may be derived from the current activities of that user, such as typing on the computer or talking on the phone. Finally, context information is *profiled* when gathered from a static source, such as a database, possibly provided directly by the user.For example, a user may indicate his/her preferences directly into the system.

*Figure 5-39* depicts an example of measurement datatype of a relational context type, namely DeviceAvailabilityMeasurement. The DeviceAvailability relational context provides the devices (attribute devices of class DeviceAvailabilityMeasurement) that are available to a person (attribute personID). In such example, the *precision* quality parameter does not apply.

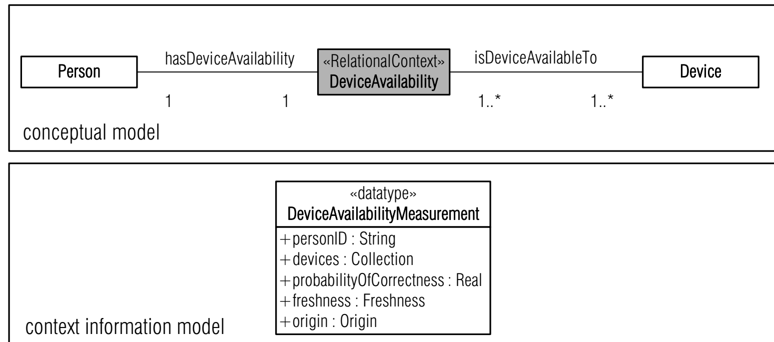



*Figure 5-39* DeviceAvailability Measurement datatype in the context information model

Figure 5-40 depicts the measurement of a situation type (SituationAvailable). The precision quality parameter does not apply to situation types, since they either hold or not. In addition, there is no need to specify the origin of a situation, since situations are always derived from other contexts or situations.

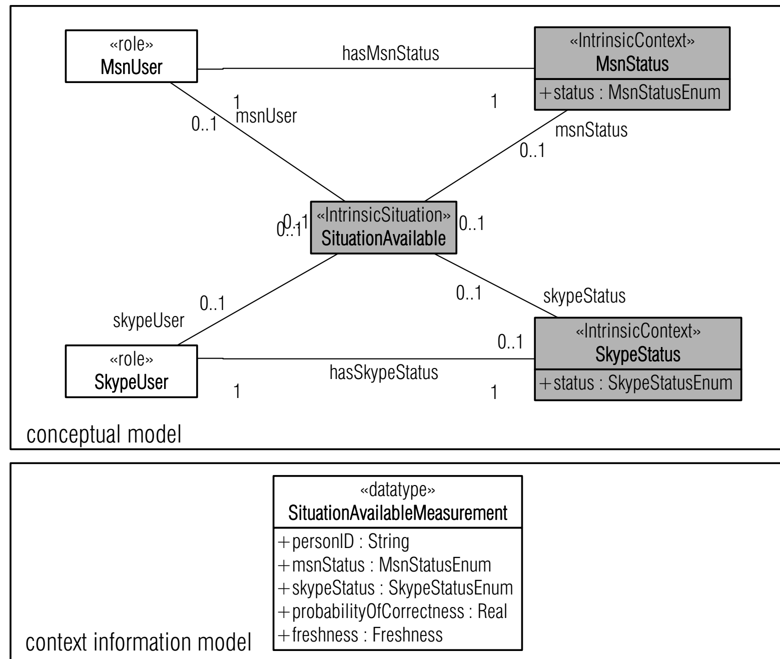



*Figure 5-40* SituationAvailable Measurement datatype in the context information model

The probability of correctness of a situation should be calculated based on the quality parameters of the elements that compose this situation, and the quality of the situation detection algorithm itself. In the example above, for instance, if information about MsnStatus or SkypeStatus is not fresh enough (e.g., freshness above 30 minutes), it is likely that the probability of correctness of SituationAvailable is quite low. Identifying such dependencies is the responsibility of the context processor designer.

### 5.6.2 Context provisioning services

In chapter 4 we have specified the interfaces of context provisioning services using opaque types, since we had not yet defined specific context and situation measurement types. In 5.6.1 we have discussed how measurement datatypes can be defined. *Figure 5-41* depicts the specification of an interface of a context source that encapsulates a GPS device. This context source supports the GeoLocationMeasurement data type, defined in *Figure 5-38*.

| «interface» GPSContextSource |
|---|
| +*subscribe(in characterization : GeoLocationMeasurement, in subscriber : SubscriberIdentification) : SubscriptionIdentification* |
| +*unsubscribe(in subscriptionId : SubscriberIdentification)* |
| +*query(in expression : GeoLocationMeasurement) : GeoLocationMeasurement* |

*Figure 5-41* Example of GPS Context Source interface specification

In order to subscribe to geographical location information, the requester should provide the characterization object (of type GeoLocationMeasurement) with the filtering attributes. For example, if the requester would like to subscribe to receive John's location, the parameter characterization should provide John's identification. For that, the value of characterization.entityID would be assigned by the requester with a unique identifier of John.

The values assigned for the quality parameters serve as minimum requirements for the filtering process, for example, if the requester would like to be notified of John's location, only when the probability of correctness of this location information is greater than 80%. This would be represented by assigning 80 as the value of the characterization.probabilityOfCorrectness attribute. The omission of attributes signifies that there are no filtering requirements for those attributes.

A query expression is similar to a subscription characterization. The filtering expression is represented by the list of attribute values of the expression object (of type GeoLocationMeasurement). Contrary to the subscription operation, a query operation returns a GeoLocationMeasurement object, which respects the filtering requirements passed as argument to the query operation.

*Figure 5-42* depicts the specification of a SituationAvailableContextManager interface, which is used by a context manager component that provides information about SituationAvailableMeasurement objects (*Figure 5-40*).

| «interface» SituationAvailableContextManager |
|---|
| +*subscribe(in transition : Transition, in characterization : SituationAvailableMeasurement, in subscriber : SubscriberIdentification) : SubscriptionIdentification* |
| +*unsubscribe(in subscriptionId : SubscriptionIdentification)* |
| +*query(in expression : SituationAvailableMeasurement) : Collection* |

*Figure 5-42* Example of SituationAvailable Context Manager interface specification

The subscribe operation of this context manager differs from the previous example, since it includes the situation state transition in which the notifications should be carried out. For example, a requester may want to be notified when the SituationAvailable of user John no longer holds. This would be specified by the state transition enterFalse. Similarly, if a requester would like to be notified when the user is available (situation begins to hold), the state transition enterTrue would be passed as argument.

The parameters characterization and expression, both of type SituationAvailableMeasurement, specify the filtering arguments. For example, if the requester would like to query all instances of SituationAvailable in which a particular user is involved (regardless of the user's status), we would assign the user's unique identifier to the expression.situationAvailable.PersonID attribute.

The result of a query operation is a collection of instances of SituationAvailable that satisfy the filtering expression passed as an argument.

Not all events of interest can be modelled (or should be modelled) as situation events. For example, consider an epileptic alarm that is detected by means of a complex algorithm that captures variations in the patient's vital signs. Due to the high complexity of the algorithm, it is not the intention to model it with the situation modelling approach proposed. An abstraction of the algorithm could be to detect the epileptic alarm by means of *primitive events*. A primitive event is a happening of interest that is not detected by means of situation transitions. Another example in which using primitive events would be appropriate is when presence information is detected by means of RFID, as opposed to using some formal relation such as nearness or distance. In such example, one would like to detect when the RFID tag is read, rather than detecting other context or situation condition to infer the presence.

We define that primitive events are generated by context source components. In order to allow exchange of primitive event notifications, event notification datatypes should be defined. For example, for the epileptic alarm, we can define the EpilepticAlarmNotification datatype, which notifies subscribers when an epileptic seizure is detected. Similarly, we can define the RFIDPresenceNotification datatype, which notifies subscribers when a particular RFID tag is read. *Figure 5-43* depicts these event notification datatypes.

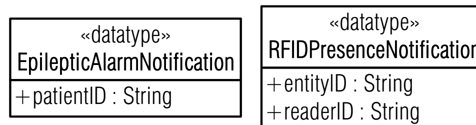



*Figure 5-43* Examples of primitive event notification structures

A context source that provides such types of event notifications is similar to the context source depicted in *Figure 5-41*. These types of context sources should provide a subscription interface that allows subscribers to be notified of primitive events. In addition, these interfaces should allow filtering of event notifications of interest. *Figure 5-44* depicts the specification of a EpilepticAlarmContextSource interface. The mechanism for filtering primitive event notification is similar to the mechanism presented previously for filtering situations and contexts.

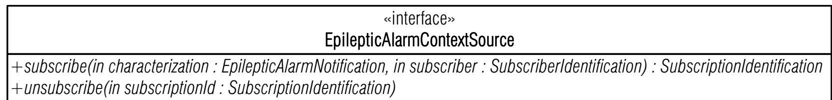



*Figure 5-44* Context source offering EpilepticAlarm event notifications

## 5.7 Discussion

We have defined in this chapter our foundational context modelling concepts, which can be used beneficially for defining application-specific or domain-specific context models. Our context foundations support the basic distinction between the concepts of *entity* and *context*, which are classified as substantials and moments, respectively.

With respect to situations, we have introduced a modelling approach based on UML and OCL constraints. This allows application designers to specify a wide range of situation types. The OCL constraints allow specification of situation types at different levels of generality. For example, it is possible to define a general situation type, such as “person has fever”, or more specific ones, such as “John and Alice have fever”.

Situations themselves can be composed of situations. This allows modularization of the situation models, improving organization and reuse of situation specifications. We have introduced situation chronoids by means of initial and final times. This allows us to explicitly capture past and present situations.

We have discussed the context information modelling phase, which considers *quality of context*. Quality of context is meta-information that describes the quality of the context information. We have discussed three quality of context parameters, namely, precision, probability of correctness and freshness. Furthermore, we have defined measurement datatypes which are used in the definition of context sources and managers interfaces.

Chapter 6

# Situation Realization

This chapter demonstrates the *feasibility* of our situation modelling approach by proposing a possible situation realization alternative. In Chapter 5 we have proposed abstractions for the specification of context-aware applications, in particular those related to the detection of situations based on context information. These context and situation abstractions facilitate context-aware application design by providing application developers with means to efficiently structure the application's state-of-affairs. This chapter proposes an approach to the *realization* of context and situation detection for *attentive* context-aware applications. Attentiveness regards the ability of application to take initiative as a result of continuously-running context reasoning activities. In order to detect situations attentively, we propose a rule-based approach to situation detection. This solution is based on the use of a general-purpose rule-based platform, which guarantees the efficiency of situation detection. The rule-based approach allows situation detection based on triggers as opposed to query-based solutions.

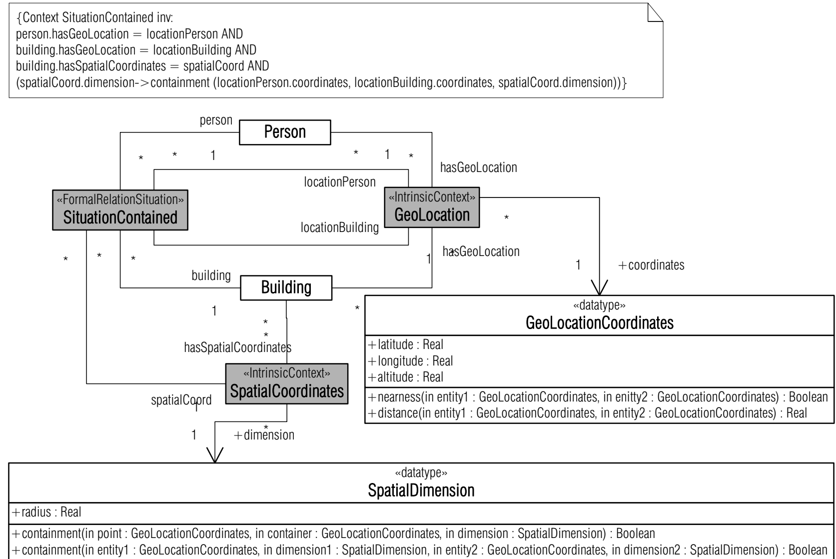
This chapter is further structured as follows: section 6.1 gives an overview of our realization approach; section 6.2 discusses background information on rule-based systems, and more specifically on Jess; section 6.3 discusses the proposed rule-based realization solution; section 6.4 elaborates on the systematic mapping between the UML class diagrams and the Java language, and between OCL expressions and the Jess language; section 6.5 discusses issues related to situation realization distribution, and finally section 6.6 presents some conclusions and important remarks.

## 6.1 From Situation Specification to Situation Realization

As discussed in chapter 5, our situation specification approach is based on UML class diagrams enriched with OCL constraints. *Figure 6-1* depicts an example of a *formal relation situation* type specification. SituationContained

specifies a situation type in which a person is inside a building. This specification is based on a context model that has been previously defined in Chapter 5, *Figure 5-27*. In this model, we have defined the entity types (Person and Building), the context types (GeoLocation and SpatialCoordinates), and formal relations (Containment), which are the building blocks for constructing the SituationContained specification.

*Figure 6-1*
SituationContained specification




The formal relation Containment relates the intrinsic context values (location values) of two spatial entities, namely, it checks whether an entity is physically contained in another entity, which is a container entity. In this particular example, the contained entity is a person, and the container entity is a building.

The OCL invariant constrains instances of SituationContained class to be associated with a person that is contained in a building. Therefore, according to this invariant, for all persons located inside a building, an instance of SituationContained is created. It would be also possible to further restrict this constraint by including, for example, the identification of the person and the building for which a situation is to be created.

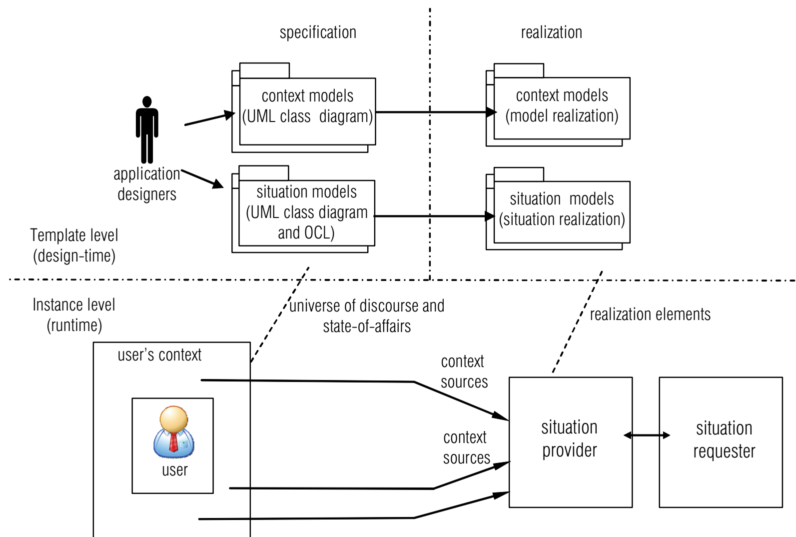
In this example we have used the *specification artefacts* for situation specification offered by our approach, namely the context models (UML classes), and the situation models (UML classes and OCL). These artefacts are used by application developers at the beginning of the design process to restrict their universe of discourse, and to define the state-of-affairs of interest, respectively. Following the design process, application developers need to choose the *realization artefacts* for situation realization. In this

Chapter we identify possible realization artefacts and we discuss how they can be used to derive realization elements.

*Figure 6-2* explicitly separates the elements of specification and realization, and highlights the correspondences between them. Context models at specification phase correspond to model realization elements in the realization phase. Similarly, situation models in the specification phase correspond to situation elements at realization phase.

*Figure 6-2* also depicts the relations between the user's context and the component-based implementation at the instance level (runtime). Context source components provide context information, which is input to situation provider components. Situation provider components have been defined in the previous chapters as context manager components.

*Figure 6-2* Correspondences between UML specifications and realization elements




Various realization alternatives could be used to implement situations, which would lead to different realization artefacts. A solution based on rules appears naturally given the nature of situation detection, in which the user's context is continuously monitored in order to check whether certain conditions (situation specification) are met. In a rule-based implementation, the designer defines rules which are repeatedly applied to a collection of facts in a working memory.

As opposed to procedural solutions, rule-based solutions offer flexibility with respect to the maintainability of the rules. Rules can be modified and added at application runtime with no need for code recompilation. Since situation specifications may change over time, and new situation specifications may be defined at application runtime, it is beneficial to use a mechanism that offers such flexibility for situation realization. For these reasons, we have based our situation realization artefacts on rule-based

systems. In the next sections we discuss rule-based systems in general and our rule-based solution in detail.

## 6.2 Rule-Based Systems

### 6.2.1 Basic concepts

Rule-based systems aim at solving domain problems by using a knowledge base expressed in terms of *rules*, which are repeatedly applied to a collection of *facts*. These two concepts are essential to a rule-based system:

- *Facts* represent circumstances that describe a certain situation in the real world; and,
- *Rules* represent heuristics that define a set of actions to be executed in a given situation.

Rules are similar to *if-then* statements of traditional programming, in which the *if* part is often called *the left hand side (LHS)*, *predicate* or *premises*, and the *then* part is the *right hand side (RHS)* or *conclusions*. The LHS consists of an expression, which can be a single expression or a combination of expressions (composite expression). For the rule to be applicable, i.e. to execute the RHS, or to derive the conclusions, the LHS should be true.

A single LHS expression is called a *pattern*[3]. A pattern is not a fact. It is an expression that defines the characteristics of a fact to be *matched*. A match occurs when there are facts that fulfil the characteristics defined in a pattern. A composite LHS expression consists of several single expressions connected together by using *conditional elements*, such as "and, or, not" in order to create complex rules. The LHS of a rule is true when the (composite) patterns in the LHS are successfully matched.

The RHS of a rule consists of a sequence of actions, typically represented by function calls, to be executed when the rule is applicable, i.e. when the expression built with patterns and conditional elements is successfully matched against facts. In addition to function calls, the RHS of a rule can also generate new facts. The general structure of a rule is the following:

If $<(\text{pattern}_1) \ldots (\text{pattern}_N)>$ then $<(\text{action}_1) \ldots (\text{action}_M)>$

---

[3] There is an essential difference between the concept of pattern just introduced, and the pattern concept discussed in Chapter 3 (architectural patterns). Architectural patterns refer to a technique used to describe a particular recurring design problem and present a generic scheme for its solutions. The term "pattern" in the sense of rules refers to the single expressions referring to facts in the LHS of a rule

In order to choose a particular rule-based technology, we have carried out a survey [21] in which we compare several rule-based solutions currently available, namely Mandarax [71], CLIPS [19], JDrew [65] and Jess [64]. Some of the parameters used in the comparison were documentation, clarity, expressiveness, platform portability, and licensing.

According to the results obtained in this research, we have concluded that Jess is the best technology for our situation realization, since (i) Jess offers free licensing for academic use, (ii) it is highly integrated with the Java platform, which enables, among others, platform portability, (iii) it is based on the Rete algorithm [44, 42], which efficiently matches the patterns for situations; (iv) it offers extensive online documentation; (v) it provides the level of expressiveness required for situation specification; and (vi) it offers JessDE [64], which provides an implementation environment integrated with Eclipse [34].

### 6.2.2 Jess

Jess (Java Expert System Shell) is a rule engine for the Java platform, which supports the development of rule-based applications that can be tightly coupled to code written in the Java language. Jess is also a powerful Java scripting environment, from which it is possible to create Java objects, call Java methods, and implement Java interfaces without compiling any Java code.

In Jess there are three ways to represent knowledge; by using the rule-based, object-oriented and procedural programming paradigms. It is possible to develop applications using only rules, only objects, only procedures, or a mixture of rules, objects and procedures. In addition, Java applications built using Jess are able to reason using knowledge supplied in the form of declarative rules.

#### *The Jess architecture*

*Figure 6-3* depicts the Jess architecture, which is composed by a *working memory*, a *rule base*, a *inference engine* and an *execution engine*. The dashed line groups the components that manipulate rules.

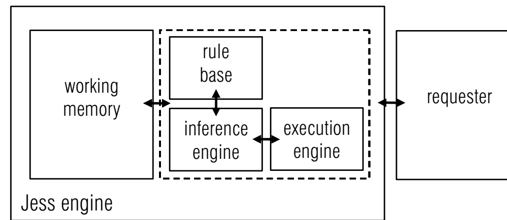



*Figure 6-3* The Jess architecture

The Jess rule engine does not contain any facts or rules until they are loaded, respectively, into the *working memory* and *rule base*. The working memory contains facts and it is sometimes also called a *fact base*. The Jess working memory is similar to a relational database, where facts are like rows of the database. Facts are maintained with indexes to speed up searches in the working memory.

The rule base contains the rules currently available to the engine. Jess holds a rule compiler for transforming rules into a format that the inference engine can manage more efficiently. Particularly, the Jess's rule compiler builds a Rete Network [44], which is a complex and indexed data structure for speeding up rule processing. Rules can only react to additions, deletions, and changes of facts in the working memory.

The *inference engine* decides what rules to *fire* and when. Firing a rule means executing the actions specified in the rule's LHS. The main subcomponent of the inference engine is the *pattern matcher,* which decides which rules to activate based on the current content of the working memory. A rule is activated when the pattern matcher finds facts that satisfy the LHS of this rule. This is not a trivial task if we take into account that the working memory may contain thousand of facts, and the rule base may contain complicated rules with several premises and conclusions. In these cases, the pattern matcher might need to search through millions of combinations of facts to find combinations that satisfy rules. This process is optimized by a rule application mechanism that is based on the Rete algorithm, which efficiently matches the patterns for situations by remembering past pattern matching tests. Only new or modified facts are tested against the rules. By using the Rete algorithm, the Jess engine can execute many orders of magnitude faster than an equivalent set of traditional *if-then* statements in procedural programs.

Finally, the *execution engine* executes the RHS of the rule once the inference engine decides what rule to fire. The Jess rule engine works in discrete cycles consisting of three steps:

- All rules contained in the rule base are compared to the working memory in order to decide which ones should be activated during this cycle. The list of activated rules, together with any other rules activated in previous cycles, is called the *conflict set*;

- The conflict set is ordered to form the agenda. This process is called conflict resolution. The used strategy depends on many factors, some of them under the programmer's control;
- To complete the cycle, the first rule on the agenda is fired and the entire process is repeated again.

***Facts and rules***

In Jess, the structure of a fact is defined by a *template* in the same way the structure of an object is defined by a class in *object-orientation*. The template has a name and a set of *slots*, i.e. a template defines the structure of a fact. As an example of template, consider the following:

```
(deftemplate car "some car" (slot model) (slot color)).
```

A fact of this template could be (car (model Ferrari) (color red)). In order to include and delete facts from the working memory, the assert and retract commands are used, respectively. Jess supports various types of facts, such as *ordered*, *unordered* and *shadow* facts. We are particularly interested in shadow facts, which are explained in detail in the next section.

For defining rules, Jess offers the defrule construct. Its general syntax is as follows:

(defrule name-of-the-rule "comment" ($pattern_1$) … ($pattern^N$) => ($action_1$)...($action^M$))

A simple example of rule that fires when a car of type Ferrari is found in the working memory is the following:

```
(defrule identify-Ferrari
(car (model Ferrari))
=>
(printout t "Ferrari found." crlf))
```

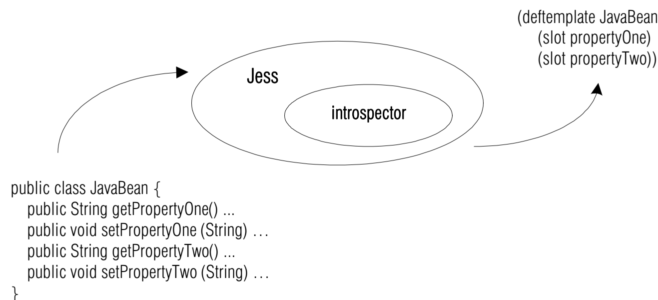
The LHS consists of the pattern (car (model Ferrari)), and the RHS consists of the function call printout, which prints a message on the default system output. If the pattern matcher component finds a car model Ferrari in the working memory, we say that there is a *match*, and the RHS should be executed. As can be seen in this example, not all structural elements (slots) of a fact have to be present in a pattern. A pattern describes the minimum characteristics for a fact to be matched. Complex combinations of patterns can be used in the LHS of a rule. We illustrate this by means of examples throughout this chapter.

### *Shadow facts*

Shadow facts are a special kind of Jess facts that serve as bridges to Java objects. They form the mechanism offered by Jess to include Java objects into the working memory. The slots of a shadow fact correspond to the properties of a *JavaBean*, which is a Java object whose properties are only manipulated through *get* and *set* methods. The get method of a JavaBean reads the value of a property, and the set method changes this value. Therefore, the properties of a JavaBean are always private, and, for each property, a pair of methods (get and set) is required. The Jess command defclass creates a template for a shadow fact, and the definstance command creates an individual shadow fact.

*Figure 6-4* (from [42]) depicts the mechanism used by Jess to realize shadow facts. The JavaBeans API offers a class called *introspector* that can inspect the beans and find properties based on the get and set methods. Jess uses this introspector to generate the deftemplate for shadow facts.

*Figure 6-4* JavaBeans and Shadow facts




The following code illustrates a simple example of a JavaBean:

```
public class Person {
    private String name = "";
    public String getName() {return name;}
    public void setName(String nm) {
        name = nm;
    }
}
```

If we declare (defclass person Person) in Jess, a deftemplate corresponding to the Person class is automatically created. In order to create an instance of Person in the working memory, we would perform (definstance person (new Person)).

Changes in the JavaBean's properties can be automatically perceived by Jess by means of a special kind of *event*. The JavaBean can inform Jess of changes in the object by sending an *event notification*, which is expected at

any time by the Jess engine. A cycle of pattern matching is performed every time an event notification is received by the Jess engine.

This characteristic is essential when dealing with attentiveness in context-awareness. Since context is quite dynamic, object attributes are continuously updated, which should be automatically reflected in the working memory. In addition, rules should be checked at the same rate as the rate in which the working memory is updated. For example, if *location* is a property of class Person, this property should have the location information of the person as up-to-date as possible. This might imply, for example, that this property has to be updated every five seconds. The property is then automatically updated in the working memory by means of event exchanges between Java and Jess. If there are rules that work on the location of a Person, this rule should be checked again every time the location of the person is changed.

## 6.3 Rule-Based Situation Realization

*Figure 6-5* refines *Figure 6-2* by considering Java and Jess as technologies for situation realization. It depicts the elements of our approach and the correspondences between the UML specification, the Java code and the Jess code at the template level (design-time). The *context model realization* elements depicted in *Figure 6-2* correspond to the actual realization as Java classes and Jess templates. The *situation realization* elements appear in the realization phase as Java classes, Jess templates and rules.

In *Figure 6-2*, at the instance level (runtime) we have depicted the situation provider, which is the component responsible for detecting situations. Since the situation detection implementation is based on Jess, the situation provider component is implemented by the Jess architecture, as depicted in *Figure 6-5*. *Figure 6-5* also shows the relations between the user's context and the rule-based implementation. The Jess architecture is simplified to improve clarity. Context sources provide context information, which is stored as facts in the engine's working memory. In the inference engine, rules of the rule base are matched against the facts of the working memory.

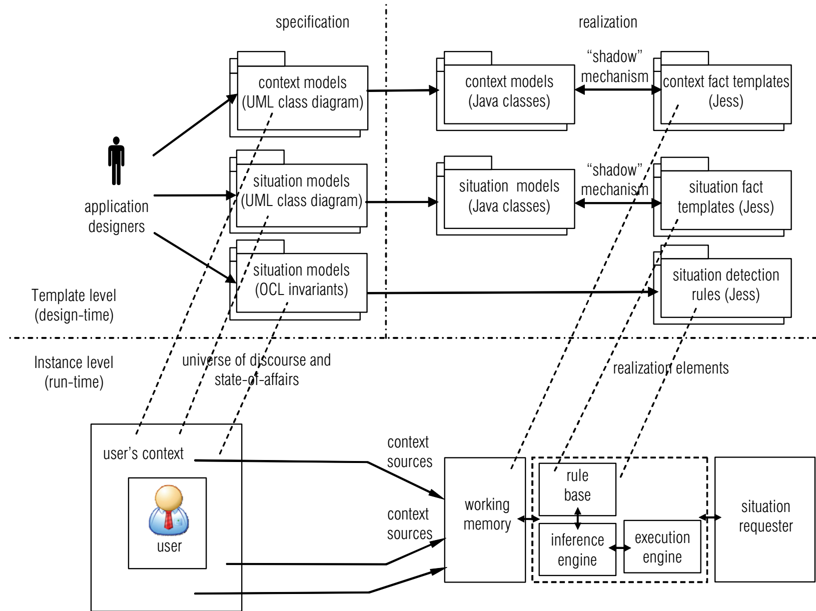



*Figure 6-5* Correspondences between UML specifications, and Java and Jess code

We have used *shadow facts* to implement our structural context models. As already mentioned, this is the mechanism offered by Jess to integrate a Java application with the working memory. Context and entity objects created in the Java application are reflected in the working memory. Therefore, any alteration of these Java objects is automatically perceived by the Jess working memory. The Java classes in our implementation directly reflect the UML models defined in the context model, such that their generation can be automated.

Similarly to the structural context model, each situation type, as specified in the UML class diagram, corresponds to a Java class, as well as to a shadow fact template. Situation instances are represented as shadow facts, called here *situation facts*, which are created and deactivated by rules for situation detection.

Once we have defined the structural and situation context models, we can carry out the situation detection realization. Each situation type generally leads to the definition of two rules, namely a rule for situation fact creation, and a rule for situation fact deactivation. Conditions for enabling these rules are derived from the OCL invariants of situation classes. The rule for situation creation detects when an invariant becomes true, and the rule for situation deactivation detects when the invariant becomes false. We have identified patterns of situation types that are systematically mapped to Jess code (see section 6.4.2).

### 6.3.1 Situation life cycle

The situation fact life cycle consists of creation, activation, deactivation and destruction. The *activation* of a situation fact occurs simultaneously to its *creation*. When the invariant holds and the situation fact does not exist yet, the situation fact is created. The *deactivation* of a situation fact occurs when the situation invariant no longer holds. *Figure 6-6* uses a UML 2.0 activity iagram to show when situation facts should be created or deactivated.

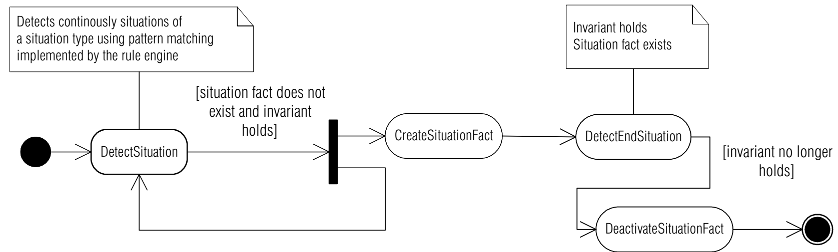



*Figure 6-6* Activity diagram for situation creation and deactivation

The activities illustrated in *Figure 6-6*, namely DetectSituation, CreateSituationFact, DetectEndSituation and DeactivateSituation are implemented in Jess by means of creation and deactivation rules. Creating a situation fact implies instantiating a situation class in the Java virtual machine, which is shadowed in the working memory by a corresponding situation fact.

Deactivated situation facts consist of historical records of situation occurrence, which may be used to detect situations that refer to past occurrences. Currently, we implement a simple rule-based time-to-live mechanism for historical records, which considers the final time of deactivated situation facts to delete an old record.

We have identified that situation realization in Jess follows certain patterns of implementation. *Table 6-2* depicts how creation and deactivation rules should be formulated in the Jess language.

*Table 6-2* Creation and deactivation rules

| Creation rule | Deactivation rule |
|---|---|
| (situation type invariant) | (not (situation type invariant)) |
| (not (situation exists)) | (situation exists) |
| => | => |
| create (situation) | deactivate (situation) |
| [RaiseEvent()] | [RaiseEvent()] |

The condition part of a creation rule checks whether the OCL invariant holds, and whether an instance of that particular situation is already currently active (final time not nil). If these conditions are met, a situation fact is created, and optionally, an event can be raised. Analogously, the

condition part of a deactivation rule checks whether the OCL invariant no longer holds, and a current situation fact active for this situation. When these conditions are met, this situation instance is deactivated, and optionally, an event can be raised.

### 6.3.2 Temporal aspects

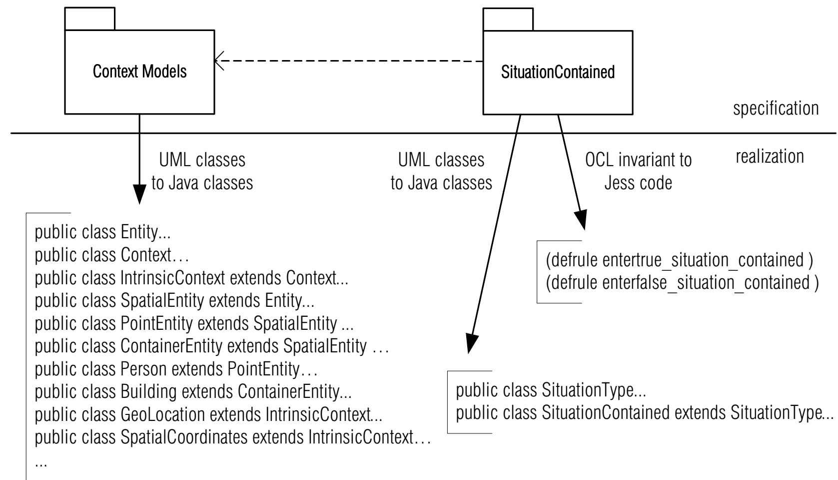
Temporal aspects are represented in our approach by means of initial and final time attributes. Each situation type extends the SituationType class and inherits the temporal attributes of this class. The initialtime attribute captures the moment a situation begins to hold, and the finaltime attribute, the moment a situation ceases to hold. Since we capture the finaltime, our model is capable of representing past occurrences of situations. When an instance of class SituationType is created, the initialtime attribute is set to the creation time.

The SituationType class also implements a deactivation method that sets the finaltime attribute, and creates serialized copies of the situation and its dependencies, which are stored for future use. Serialization is necessary to preserve the situation state at the time the situation was deactivated. Serialized SituationType objects are given unique identifiers, so that they can be retrieved unambiguously.

### 6.3.3 Containment example

Consider the SituationContained specification presented in *Figure 6-1*. This situation specification is based on a context model depicted in *Figure 5-27* in Chapter 5. As we have discussed before, context models correspond to Java classes in the realization phase. Similarly, situation UML classes correspond to Java classes, and the OCL constraints correspond to Jess rules. *Figure 6-7* depicts an overview of the correspondences between UML and OCL specifications, and Java and Jess code for the SituationContained example.

*Figure 6-7* Correspondences between UML and Java, and between OCL and Jess, for the SituationContained example




The Java classes are generated almost literally from the UML specifications. For example, the UML classes Entity and Context, and their association correspond to the Java classes Entity and Context and their attributes in the realization phase. The same applies to the other classes of our context and situation models. Jess rules are also systematically derived from the OCL constraints. *Figure 6-8* depicts the EnterTrue Jess rule derived from the OCL constraint of the SituationContained class.

*Figure 6-8* SituationContained EnterTrue Jess rule

```
(defrule entertrue_situation_contained
  (Person (OBJECT ?person)( hasGeoLocation ? person_hasGeoLocation))
  (GeoLocation (OBJECT ?locationPerson&:(eq ?locationPerson ? person_hasGeoLocation)))
  (Building (OBJECT ?building)(geoLocation ?building_hasGeoLocation))
  (GeoLocation (OBJECT ?locationBuilding&:(eq ?locationBuilding ?building_hasGeoLocation)))
  (Building (OBJECT ?building)(spatialCoordinates ?building_hasSpatialCoordinates))
  (SpatialCoordinates (OBJECT ?spatialCoord&:(eq ?spatialCoord ?building_hasSpatialCoordinates)))
  (GeoLocation (OBJECT ?locationPerson) (location ?locationPerson_coordinates))
  (GeoLocation (OBJECT ?locationBuilding) (location ?locationBuilding_coordinates))
  (SpatialCoordinates (OBJECT ?spatialCoord) (dimension ?spatialCoord_dimension))
  (test (call context_control.SpatialDimension Containment ?locationPerson_coordinates
                              ?locationBuilding_coordinates ?spatialCoord_dimension))
   (not (SituationContained (OBJECT ?st)(person ?person) (building ?building) (finaltime nil)))
  =>
  (bind ?SituationContained (new situation_control.SituationContained ?person ?building))
  (definstance SituationContained ?SituationContained)
)
```

The condition part of the entertrue_situation_contained rule checks whether there is (i) a person associated to a GeoLocation context object, and (ii) a building associated to both, a GeoLocation and a SpatialDimension context object. These parts of the condition correspond to the first lines of the OCL invariant

defined in *Figure 6-1*. The condition also checks whether the person is actually contained in the building by means of the Containment formal relation. In addition, it checks whether a SituationContained instance does not already exist for that person and building, which would mean that this situation instance had been already created. When these conditions are met, the action part is triggered, i.e. an instance of SituationContained is created for that person and building.

The EnterFalse rule is quite straightforward; it is just the negation of the invariant, and the pattern for the existence of the situation fact, as depicted in *Figure 6-9*. Contrary to the entertrue_situation_contained, the condition part of the enterfalse_situation_contained rule checks whether the person is no longer contained inside the building, and if there is an active SituationContained instance for that person and building. If these conditions are met, that particular instance of SituationContained is deactivated, and can be used in the future as a historical record.

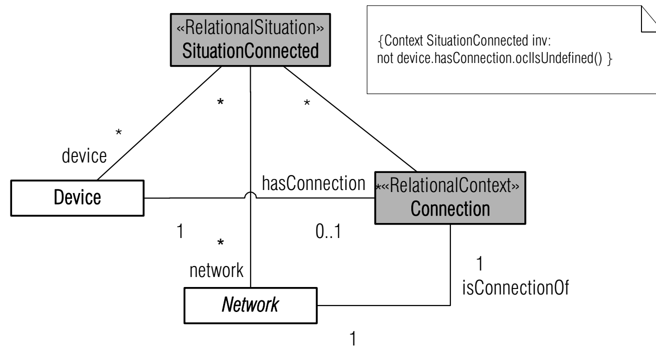
*Figure 6-9* SituationContained EnterFalse Jess rule

```
(defrule enterfalse_situation_contained
  (not (and (Person (OBJECT ?person)( hasGeoLocation ? person_hasGeoLocation))
        (GeoLocation (OBJECT ?locationPerson&:(eq ?locationPerson ? person_hasGeoLocation)))
        (Building (OBJECT ?building)(geoLocation ?building_hasGeoLocation))
        (GeoLocation (OBJECT ?locationBuilding&:(eq ?locationBuilding ?building_hasGeoLocation)))
        (Building (OBJECT ?building)(spatialCoordinates ?building_hasSpatialCoordinates))
        (SpatialCoordinates (OBJECT ?spatialCoord&:(eq ?spatialCoord ?building_hasSpatialCoordinates)))
        (GeoLocation (OBJECT ?locationPerson) (location ?locationPerson_coordinates))
        (GeoLocation (OBJECT ?locationBuilding) (location ?locationBuilding_coordinates))
        (SpatialCoordinates (OBJECT ?spatialCoord) (dimension ?spatialCoord_dimension))
        (test (call context_control.SpatialDimension Containment ?locationPerson_coordinates
                                        ?locationBuilding_coordinates ?spatialCoord_dimension))
   (SituationContained (OBJECT ?st)(person ?person) (building ?building) (finaltime nil))
  =>
  (call ?SituationContained deactivate)
)
```

### 6.3.4 Wireless network connections example

*Figure 6-10* and *Figure 6-11* depict other examples of situation specification, namely SituationConnected and SituationSwitch specifications, which have been already discussed in Chapter 5. SituationConnected specifies a situation type in which a device has established a connection (relational context type) to each of the two network types WLAN and Bluetooth (entity types). SituationSwitch refers to the situation in which a device switches from a WLAN connection to a Bluetooth connection. This situation may be used to set, for example, new quality of service parameters.

*Figure 6-10* SituationConnected specification




The OCL invariant of SituationConnected (*Figure 6-10*) defines that instances of this situation must be associated with at least one connection object. SituationSwitch (*Figure 6-11*) is modelled by composing multiple occurrences of SituationConnection, one called wlan, and the other called bluetooth. In order to detect SituationSwitch, we would have to verify whether SituationConnected held in the past for network WLAN, and currently holds for network Bluetooth. This requires the use of temporal aspects, which are described in the OCL invariant in terms of initial and final times. We have defined the constraint that the handover time should not be longer than one second. In addition, in order to avoid that the active instance of SituationSwitch endures indefinitely, we have included the condition in which the time interval between now (current time) and the initial time of SituationConnected for network Bluetooth should not be bigger than two seconds. This allows instances of SituationSwitch to be active for some time, but not for more than three seconds.

*Figure 6-11* SituationSwitch specification

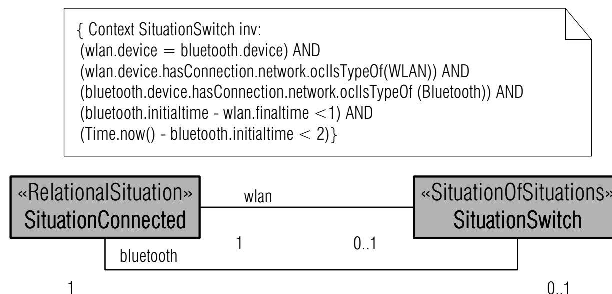



*Figure 6-12* depicts an overview of the correspondences between UML and OCL specifications, and Java and Jess code for the SituationConnected and SituationSwitch example. Examples of Java classes in the realization are Device, Network and Connection, which correspond to classes of the same name in the UML class specifications. The same applies to the other classes of our

context and situation models. The enter_true and enter_false Jess rules are also systematically derived from the OCL invariant.

*Figure 6-12* Correspondences between UML and OCL, and Java and Jess code for the SituationConnected and SituationSwitch example

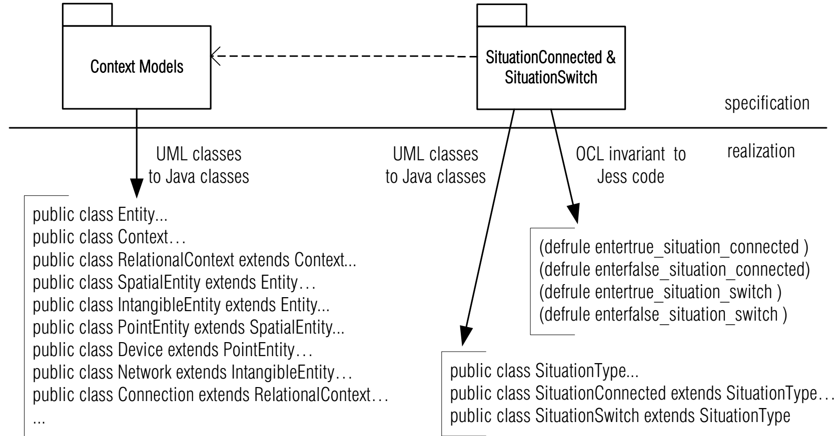



*Figure 6-13* depicts the Jess rules derived from the OCL constraint of the SituationConnected class. The condition part of the entertrue_situation_connected rule checks whether there is a Connection relational context in the list of contexts of that device. This part of the condition corresponds to the OCL invariant defined in *Figure 6-10*. In addition, it checks whether a SituationConnected instance does not already exist for that device. When these conditions are met, the action part is triggered, i.e. an instance of SituationConnected is created for that device.

The condition part of the enterfalse_situation_connected rule checks whether the device is no longer connected to a network, and if there is an existing SituationConnected for that device. If these conditions are met, that particular instance of SituationConnected is deactivated, and can be used in the future as a historical record.

*Figure 6-13* SituationConnected realization in Jess

```
;EnterTrue (SituationConnected)
(defrule entertrue_situation_connected
   (Device (OBJECT ?device) (hasConnection ?hasConnection))
   (not (SituationConnected (OBJECT ?situation) (device ?device) (finaltime nil)))
    =>
   (bind ?SituationConnected (new situation_control.SituationConnected ?device))
   (definstance SituationConnected ?SituationConnected))

;EnterFalse (SituationConnected)
(defrule enterfalse_situation_connected
   (not (Device (OBJECT ?device) (hasConnection ?hasConnection)))
   (SituationConnected (OBJECT ?situation) (device ?device) (finaltime nil)))
   =>
   (call ?situation deactivate))
```

*Figure 6-14* depicts the Jess rules derived from the OCL invariant of the SituationSwitch class. The condition part of the entertrue_situation_switch rule checks whether there was an instance of SituationConnected associated to network WLAN in the past (finaltime not nil), and currently there is an instance of SituationConnected associated to network Bluetooth. In addition, the handover time should not be longer than 60 seconds, and the time interval between the SituationConnected for network Bluetooth starting time and the current time (call System currentTimeMillis) should not be bigger than two seconds. These parts of the condition correspond to the OCL invariant depicted in *Figure 6-11*. As in all creation rules, the condition also checks whether there is no instance of SituationSwitch for that particular currently active handover. When these conditions are met, an instance of SituationSwitch is created.

*Figure 6-14* also depicts the enterfalse_situation_switch, which checks whether there is a situation switch instance currently active with time interval between the SituationConnected for network Bluetooth starting time and the current time bigger than two seconds. If these conditions are met, the active situation instance is deactivated.

As already mentioned in section 6.3.2, we use a mechanism based on object serialization to deactivate situations in order to preserve the situation state and the time the situation was deactivated. The identification of the serialized objects is based on unique identifiers, which are generated by the application implementation. For this reason, when checking the existence of a past instance of SituationConnected, we have used the device's unique identifier instead of the object identifier. The Jess command call invokes a method on a given object, so the command (call ?device getIdentity) invokes the getIdentity method on the device object, which returns the device's unique identifier. For currently active situation facts, we would use the Jess command (OBJECT ?device), which refers to the object identifier.

*Figure 6-14* SituationSwitch realization in Jess

```
;EnterTrue (SituationSwitch)
(defrule entertrue_situation_switch
   (Device (OBJECT ?dv) (identity ?dvid))
   (SituationConnected (OBJECT ?SWlan)
                (device ?device&:(eq (call ?device getIdentity) ?dvid))
                (network ?net&:(instanceof ?net context_control.WLAN)) (finaltime ?finaltime&:(neq ?finaltime
   (SituationConnected (OBJECT ?SBlue) (device ?dv)
                (network ?net2&:(instanceof ?net2 context_control.Bluetooth)) (starttime ?start) (finaltime nil)
   (test (<= (- (call ?start getTime)(call ?finaltime getTime)) 60000))
   (test (<= (- (call System currentTimeMillis) (call ?start getTime)) 120000))
   (not (SituationSwitch (OBJECT ?situation) (wlan ?SWLAN) (bluetooth ?SBlue)  (finaltime nil)))
   =>
   (bind ?SituationSwitch (new situation_control.SituationSwitch ?SWlan ?SBlue))
   (definstance SituationSwitch ?SituationSwitch))

;EnterFlase(SituationSwitch)
(defrule enterfalse_situation_switch
   (Device (OBJECT ?dv) (identity ?dvid))
   (SituationConnected (OBJECT ?SWlan)
          (device ?device&:(eq (call ?device getIdentity) ?dvid))
          (network ?net&:(instanceof ?net context_control.WLAN))(finaltime ?final&:(neq ?final nil)))
   (SituationConnected (OBJECT ?SBlue) (device ?dv)
           (network ?net2&:(instanceof ?net2 context_control.Bluetooth))(starttime ?start))
   (test (not(<= (- (call ?start getTime)(call ?final getTime)) 60000)))
   (test (not (<= (- (call System currentTimeMillis) (call ?start getTime)) 120000)))
   (SituationSwitch (OBJECT ?situation)(wlan ?SWlan) (bluetooth ?SBlue)(finaltime nil))
   =>
   (call ?situation deactivate))
```

## 6.4 Mappings

So far, we have identified the realization artefacts for situation realization and we have seen how the realization elements can be derived from the specification elements using these artefacts. We have not discussed yet how to systematically *map* the realization elements from the specification elements. More specifically, we have not yet discussed how to map the UML specification to Java classes, and how to map he OCL specifications to Jess rules. This section elaborates on how these artefacts can be derived from the specifications.

### 6.4.1 UML class diagrams to Java

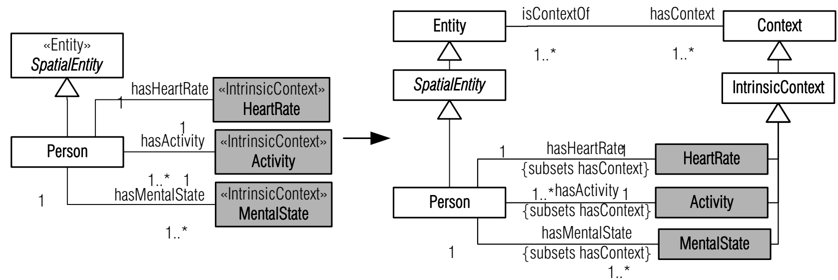
***Mapping of the model specifications***

In order to map our specification models to the Java language, we bring the context metamodel and the model levels (Chapter 5) to the same model level, so that the original relationships between metamodel classes and model classes become *specialization* relationships. This is necessary since we

cannot extend the Java language with the context profile we have proposed in Chapter 5, as we have extended UML with this profile.

*Figure 6-15* depicts an example of the transformation from the specification using the context profile (on the left), and a specification using inheritance (on the right). In the specification using a profile, classes refer to (instantiate) classes in the profile by means of stereotypes. For example, class SpatialEntity instantiates class Entity by means of the stereotype «Entity», which is defined in the context profile. In the other modelling alternative, class SpatialEntity simply extends class Entity by means of the specialization mechanism.

*Figure 6-15* Transformation to specification without context profile




Similar to classes, association ends also use different extension mechanisms in the modelling alternatives. In the modelling alternative using a context profile, although not depicted in *Figure 6-15*, association ends are also stereotyped. For example, association end hasHeartRate is stereotyped with stereotype «hasContext», meaning that this association end instantiates association end hasContext. In the modelling alternative using inheritance, we use the subsets constraint to explicitly label an association end that extends another association end. As already explained in Chapter 5, the UML constraint {subsets <end>} indicates that an association end constrains the possible values of association end <end>. For example, the possible values of association end hasHeartRate are a subset of the possible values of association end hasContext (defined between classes Entity and Context).

Regarding the mappings from UML classes to the Java language, we consider the modelling alternative using specializations in the following sections.

***Model mappings***

In general, the mapping from the model realization elements (Java classes) to the specification elements (UML classes) is straightforward: classes in the UML specifications map to classes in the Java code; hierarchy of classes in UML map to hierarchy in the Java code; and associations in UML map to attributes in Java, respecting association end naming and cardinality.

Association ends are mapped to attributes in the Java implementation, considering the cardinality. The following types of associations are supported [68]:

- *One-to-one* associations can be implemented using two attributes in the two *associated classes*, each one making a reference to the other. *Set* and *get* methods are used to set new values to these attributes, and to read the value of the attributes, respectively. This solution can also be used when one or both of the association ends is optional, that is, the cardinality of the association ends is zero or one (0..1);
- *One-to-many* associations can be implemented using two attributes in the two associated classes in which one of these attributes is a collection. Set and get methods are also used to set and read attribute values, respectively. Since one of the attributes is a collection, operations to manipulate lists are necessary, such as add/remove an object to/from the collection, and so forth;
- *Many-to-many* associations can be implemented similarly to the one-to-many association, except that the attributes on both associated classes are collections. Therefore, besides get and set operations, it is also necessary to implement methods to manipulate collections in both classes participating in the associations;
- *One-way navigable association* can be implemented in such way that the class which the arrow is pointing at does not need additional set and get methods. The other class needs an attribute, and get and set operations. In case the cardinality is many, the methods for adding and removing associated objects from the collections are necessary.

Association ends constrained by subsets are not mapped to attributes in the realization phase. These association ends are treated separately since they are actually a subset of the association ends defined between the parent classes. If the association end between parent classes is already implemented, i.e. attributes in both parent classes have been created, the children classes should not implement the association end by means of attributes again. We have elaborated a solution for implementing subsets association ends based on filtering methods that collect (from the parent association ends (attributes)), the objects that in fact belong to the children classes. The method return type depends on the cardinality of the subset association end: for cardinality *many*, a collection is returned; for cardinality *one*, a single object is returned.

Consider as an example the association end hasHeartRate between classes Person and HeartRate, which is a subset of association end hasContext. In the implementation, the association ends between classes Entity and Context are mapped to collection attributes in both classes. In the Entity class, there is an attribute hasContext, which is a collection of objects of type Context. Similarly, in the Context class there is an attribute, which is a collection of objects of

type Entity. In order to implement the hasHeartRate association end, we define a method in class Person (with name hasHeartRate) that returns the objects of type HeartRate from the hasContext collection attribute. Since the cardinality of association end hasHeartRate is one, this method returns only one object of type HeartRate, as opposed to a collection. The same rationale could be applied to the counterparts of association end hasHeartRate, namely isHeartRateOf (not depicted in *Figure 6-15*), which is a subsets of association end isContextOf. A method in the HeartRate class is created in order to return all the objects of type Person from the isContextOf collection attribute. Since the cardinality of association end isHeartRateOf is also one, only one object of type Person is returned.

Datatypes in the UML specifications are mapped to normal classes in the Java implementation. Objects that are instance of these classes do not hold state, i.e. they just represented structures of values. The formal relation operations defined as methods of datatype classes in the UML specifications are mapped to *static* methods in the Java implementation. As opposed to normal class methods, static methods can be invoked without instantiating the class they belong to. This implementation decision is in line with the concept of a formal relation discussed in Chapter 5. Formal relations operate on quality values (and not objects), which are passed as parameters for the method. Therefore, the parameters of a method representing a formal relation should always be datatypes.

Our classes in the Java implementation should be JavaBeans in order to support shadow facts, as discussed in section 6.2.2. This issue has been tackled by implementing get and set methods for each attribute. In addition, changes in the Java objects should be reflected in the Jess working memory by exchanging a special kind of event, namely a java.beans.PropertyChangeEvent. In order to support event exchange between the Java virtual machine and the Jess engine, our Java classes should offer support for a PropertyChangeListener, which enables the notification of changes from Java to Jess. Whenever an attribute is changed in the Java code, its corresponding slot value should also be changed in order to keep the Java code and the working memory synchronised. This is done by the programmer by invoking the firePropertyChange method every time an attribute value is modified.

Since we deal with situational temporal aspects, we need to use serialization to keep the state of situations and their dependencies at the time of deactivation. In order to allow serialization, all our classes implement the Serializable interface.

The majority of UML tools offer code generation facilities for the Java language [46, 60, 94]. We have used Octopus [96] for generating Java code. Octopus is able to statically check OCL constraints, and to transform the UML model, including the OCL expressions, to the Java language. Although

the mappings from the specification elements to realization elements may be automated, there are some issues that currently available automation tools do not tackle, such as the association type “subsets”. Application developers, using code generation tools need to treat subsets associations manually, or create a customized automation mechanism to process subsets in the proper way during the code generation.

### 6.4.2 OCL invariants to Jess

OCL and Jess are different languages aiming at different goals. OCL is a formal constraint language used to describe expressions on UML models. These expressions aim at specifying the modelling aspects which cannot be covered by the diagrams. Several types of OCL expressions can be formulated in a UML diagram, such as invariants, preconditions, postconditions, among others. In contrast, the Jess language aims at providing application developers with means to write rules that can be accepted and understood by the Jess engine.

Our specific goal is to map one type of OCL expression, namely *invariant*, to LHS (condition parts) of Jess rules. As we have mentioned in previous sections, an OCL invariant generates at least two Jess rules in the implementation, one to create a situation fact (*enter true* rule) and the other to deactivate a situation fact (*enter false* rule). We discuss in this section how inner parts of an invariant are mapped to specific Jess expressions.

In our approach, an OCL invariant always refer to a SituationType, which is, in UML terms, the *context* of the invariant. We use the term *reference class* here instead of context, in order to avoid confusion with the term context in the scope of context-awareness. For example, in *Figure 6-10*, the OCL invariant expression refers to instances of the SituationConnected type class, which is the reference class. In such expressions, navigating to other associated objects always starts from the objects instances of a reference class. This way, the expression device.hasConnection.oclIsUndefined() (in *Figure 6-10*), starts from SituationSwitch class, so that device is an association attribute of SituationConnected class.

In the Jess implementation, the concept of a reference class does not exist. Initially, there are no instances of situation types in the system. These instances, which we call situation facts, are created when the condition part of *enter true* rules become true. For this reason we cannot start navigating from a situation fact in the condition part of an enter true rule. Instead, Jess conditions consider a general view of the system, in which the objects that participate in an invariant are always inspected in order to check whether the invariant holds or not.

As an example, consider the invariant defined in *Figure 6-10* (device.hasConnection.oclIsUndefined), which specifies that for all instances of

SituationConnected, there must be an attribute device, which is connected to a network (hasConnection.oclIsUndefined). In Jess, this invariant maps to the condition part of a rule in which we check whether there is *any* device connected to a network. If this holds, an instance of SituationConnected class, which is associated with that particular device, is created in the working memory. The mappings from OCL and Jess discussed in this section always consider this fundamental difference between these languages.

OCL invariants are Boolean expressions, similar to LHS of Jess rules. In the outer most occurrences of the AND logical operator, each operand of the AND is mapped to a line in the LHS of a Jess rule, each line representing a pattern to be matched against the contents of the working memory. For example, if an OCL invariant defines the expression ($expr_1$) AND ($expr_2$) AND ($expr_3$) this would be mapped to three lines in the rule, namely line 1 ($expr_1$), line 2 ($expr_2$), and line 3 ($expr_3$).

Jess defines a special slot called OBJECT for each shadow fact. This slot allows one to access Java objects in the working memory. For example, in order to refer to $object_1$ in the working memory, we would define in Jess (ObjectType (OBJECT ?$object_1$)), where ObjectType refers to the type of $object_1$, or the corresponding shadow fact definition (looking from the Jess engine perspective). In Appendix A we discuss in detail the mappings for specific elements of the OCL language.

## 6.5 Distribution Issues

So far, we have focussed on the various rule patterns for the detection of the various kinds of situation. We have presented the realization solutions without regard for distribution, as if situation detection were based on a single rule engine, working with a single set of rules and a single working memory. In this section, we consider alternative distribution scenarios, and discuss their trade-offs.

### 6.5.1 Simple scenarios

In *Figure 6-16* we consider the fully centralized scenario, in which no distribution is employed. In this scenario, context sources feed context information into the central rule engine's working memory. This is the simplest scenario, and has limited scalability with respect to the number of situations detected, even when situations are entirely independent of each other, i.e. when situations are detected using context conditions that are sensed independently, and are not composed of other situations. The centralized approach introduces a single point of access to context information, which can be considered a potential (privacy) hazard, due to

the sensitive nature of particular kinds of context information. In addition, such centralized scenario can present performance and scalability problems, since situation reasoning is concentrated on a single engine.

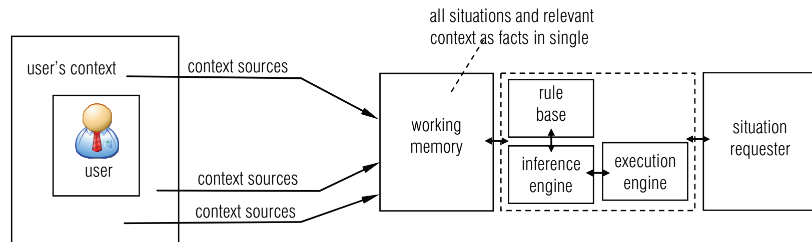



*Figure 6-16* Centralized scenario

In Figure 6-17 we consider a scenario with multiple hub-and-spokes for situation detection. In this scenario, multiple engines detect independent situations. In this approach, each rule engine may be associated to a different administrative domain, which enables more fine-grained control of the (privacy) policies which apply to the context information for that domain. The level of distribution is constrained by the nature of the situation model, each hub-and-spoke pattern consisting of a centralized solution. The solution is highly constrained by the nature of the situation model, since all related situations must be detected in the scope of the same rule engine. *Figure 6-17* depicts this solution with two rule engines detecting independent situations 1 and 2.

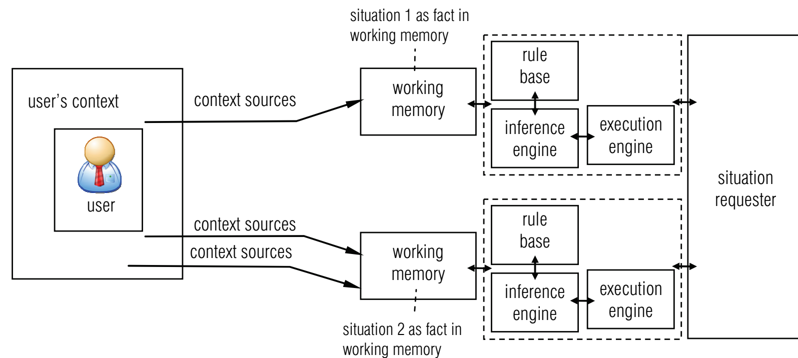


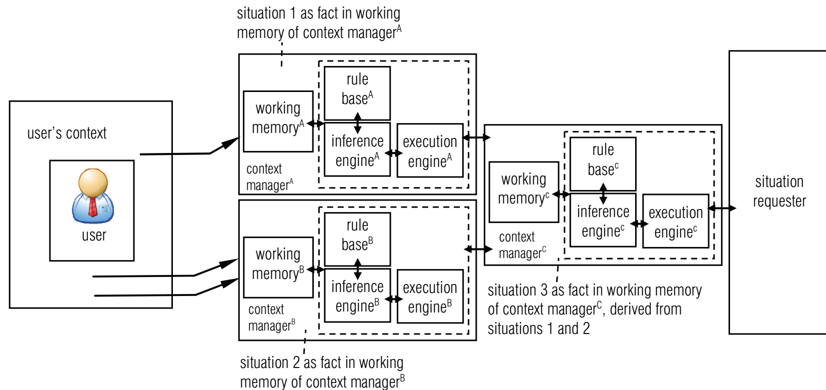

*Figure 6-17* Multiple hub-and-spokes scenario

## 6.5.2 Service-oriented architecture

We consider now a distribution scenario with a higher level of distribution that not only exploits possible independent situations, but that is able to decompose situation detection further, and distribute parts of the rule detection functionality to different rule engines. Different distribution strategies and rule engine configurations can be accommodated using this approach. *Figure 6-18* depicts a possible configuration with two independent

situations 1 and 2 detected independently in rule engines A and B (as in the hub-and-spokes scenario). The facts corresponding to those situations are shared with a rule engine C, which detects a situation 3 which is derived from situations 1 and 2. We propose a mechanism based on a service oriented architecture as one of the realization alternatives for this approach.

*Figure 6-18* Distributed scenario using service oriented approach




### *Architectural patterns*

Service oriented architectures allow us to encapsulate rule engines in components, so that situation facts are exchanged by means of situation events, which are offered through components' services. These components are the *context manager* components, which have been discussed in Chapters 4 and 5. We have presented in Chapter 5 how to design an interface for a context manager component, but we had not yet discussed the internal design of such a component, since the internal design depends on the realization alternative chosen. As we have discussed in Chapter 2, components are not forced to use any specific realization alternative, since the internal configuration of a component is not visible to other components.

Using a rule-based alternative for situation realization suggests that a context manager component should implement a rule engine for situation detection. Therefore, in *Figure 6-18*, context managers A, B and C implement rule engines A, B and C respectively. In addition, these components should be able to subscribe to other context manager components, or other context source components, in order to gather the necessary information to detect the situation of interest. In the example presented in *Figure 6-18*, situation 3 depends of situations 1 and 2 to be detected, and therefore, context manager C subscribes with context managers A and B in order to gather situations 1 and 2, respectively. In this way, a hierarchy of context manager components can be created, so that we can apply the *hierarchy of context sources and managers* architectural pattern discussed in Chapter 4. The subscription mechanism is based on the

publish-subscribe approach, which has been discussed in Chapter 2. In the next section we discuss the context manager components and the publish-subscribe mechanism in more detail.

This distributed scenario enables fine-grained control of the policies that apply to context information, since different rule engines and parts of situation detection can be associated with different administrative domains. The policies for context information may justify the usage of different distribution strategies. For example, consider an application that uses the distance between two users to determine whether users can view each other's contact information. Suppose further that GPS location is used to compute the distance between users. Due to the sensitive nature of the "raw" GPS location, different policies may apply to this information, and to the aggregate and usually less sensitive distance information. In this case, GPS location should be only available to the engines that derive proximity information. Only the aggregated proximity information should be shared with other engines that define contact information visibility.

***Context managers***

As discussed in Chapter 5, a context manager component offers a subscription interface to its users, such that situation event notifications can be delivered when these events occur. For example, the component interface specification depicted in *Figure 6-19*, offers a subscription interface for SituationContained events. The subscribe operation allows the requester to specify the situation state transition in which the notifications should be carried out. For example, a requester may want to be notified when user John enters a building, which is specified by the state transition *enter true*, or when John leaves a building, which is specified by the state transition *enter false*.

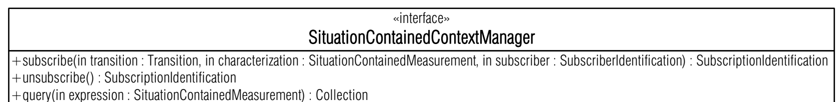
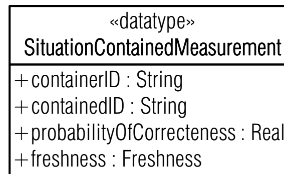



*Figure 6-19* Definition of the Situation Contained context manager component interface

The parameters characterization and expression, both of type SituationContainedMeasurement, specify the filtering arguments. *Figure 6-20* depicts the specification of the SituationContainedMeasurement datatype. This datatype refers to the SituationContained specification, which is depicted in *Figure 6-1*. SituationContained specifies a situation type in which a person is inside a building. The filtering arguments for the subscription operation can be defined at several levels of specificity. It is possible, for example, to subscribe to SituationContained events for (i) all users entering any building; (ii) user John entering any building; or (iii) user John entering a specific building. If the requester would like to be notified of events when John

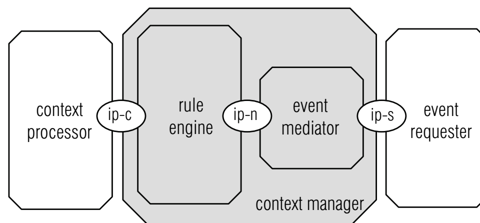
enters the "Zilverling" building, we would assign the unique identifiers of entities John to parameter containerID, and Zilverling to parameter containerID. In addition, the transition argument is assigned to enterTrue. With this type of subscription, notifications are delivered to the subscriber every time John enters the Zilverling building.

*Figure 6-20* Definition of the Situation Contained Measurement datatype




Exchanging events between event requester and provider is realized by the publish-subscribe interaction pattern. The publish-subscribe interaction pattern, as discussed in section 2.3.3, introduces the mediator role, with which events are published and subscribed. Subscribers express their interest in an event and are subsequently notified of any event, generated by a publisher, which matches their registered interest. In our architecture, a context manager component implements both the publisher and the mediator of situation events. Therefore, in addition to implementing a rule engine for situation detection, a context manager component includes an event mediator, which handles subscriptions and delivers notifications. *Figure 6-21* depicts the structure of a context manager component. The rule engine detects situation events, the event mediator mediates publish, subscribe and notify messages, the event requester subscribe to event notifications and the context processor may feed the rule engine with relevant context information necessary to detect situations.

*Figure 6-21* Structure of context manager component



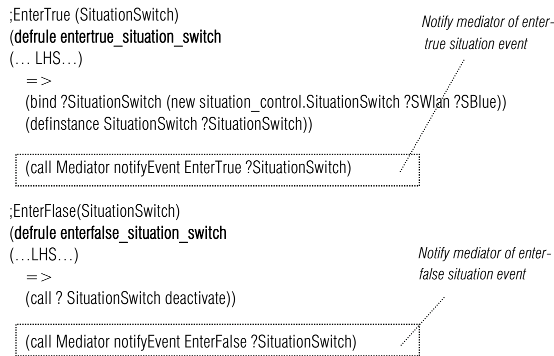

Interaction points of type ip-s allow event requesters to subscribe, query and unsubscribe to event notifications, which are the operations defined by the context manager interface, as described in *Figure 6-19*. In addition, through this interaction point, the mediator delivers event notifications to the respective requesters. The relationships between the context manager and its requesters are defined at runtime.

Interaction points of type ip-n allow the rule-engine to publish situation events with the event mediator. Contrary to the relationships between the

context manager and event requesters, the relationship between the rule engine and the mediator is defined at design-time. In order to notify the mediator that a certain situation event has occurred, an operation invocation is included in the action part of EnterTrue and EnterFalse rules. This operation calls the notifyEvent operation with the mediator component, passing as arguments the event transition (EnterTrue or EnterFalse) and the situation object.

For example, *Figure 6-22* depicts the EnterTrue and EnterFalse rules for SituationSwitch detection. In addition to activating and deactivating situation instances, these rules also notify the mediator of the occurrence of these events. The mediator then distributes these event notifications to the interested event requesters.

*Figure 6-22* EnterTrue and EnterFalse rules for situation detection – focus on notifying the mediator of situation events




Context manager components may play the role of event requester. In this case, interaction points of type ip-c allow the context manager to receive notifications from other context manager components. These notifications are included into the working memory, which is used as knowledge to derive new situation events. Context manager components may also play the role of context information requester. In this case, interaction points of type ip-c allow the context manager to receive context information from other context source components. These interactions have been discussed in section 4.5.2.

### 6.5.3 Djess

A different realization alternative for the distributed scenario uses the shared memory mechanism offered by the DJess middleware [14]. With this mechanism, rule engines running in different nodes can apply rules on shared sets of facts. A rule engine may participate in multiple shared memory partnerships (which are called Web of Inference Systems in DJess), each of which defining a shared set of facts, thus allowing arbitrary

configurations. *Figure 6-23* depicts the distributed scenario implemented using the DJess middleware.

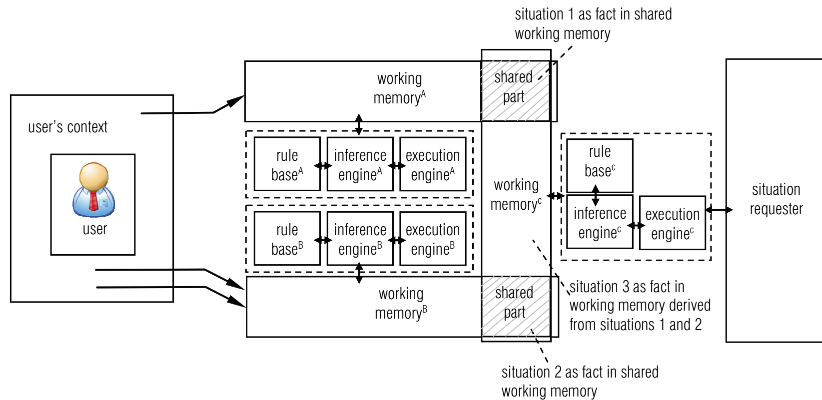



*Figure 6-23* Distributed scenario using DJess

DJess is a middleware that allows Jess engines running on different nodes of a network to communicate. Communication and coordination of these engines is achieved through the ability of each engine to transparently and asynchronously reason on knowledge located on remote nodes. Conflicts may occur when multiple nodes try to access the same shared fact by executing interfering rules at the same time. DJess addresses consistency problems due to concurrency. Consistency is guaranteed by a two-phase locking scheme, in which rule firing execution is divided into three steps: lock acquisition, rule execution and lock release.

Using DJess has advantages over a component-based solution (section 6.5.2) since it provides transparencies for situation distribution, i.e. each engine works on the distributed facts as if they were local. Therefore, there is no need to bother with subscribe/notify messages, since communication functions are hidden by DJess. However, DJess does not support shadow facts, and because of that, Java objects in the Java implementation cannot be tightly integrated with DJess working memory. In order to work in DJess, Java objects have to be asserted in the working memory as a native Jess fact, and not as a shadow fact. Furthermore, fact templates have to be manually created using the deftemplate command, as opposed to the defclass command that would automatically create a deftemplate given a class definition. In addition, contrary to the shadow facts mechanism, changes in Java objects have to be manually informed to the DJess engine using the "modify" command of Jess.

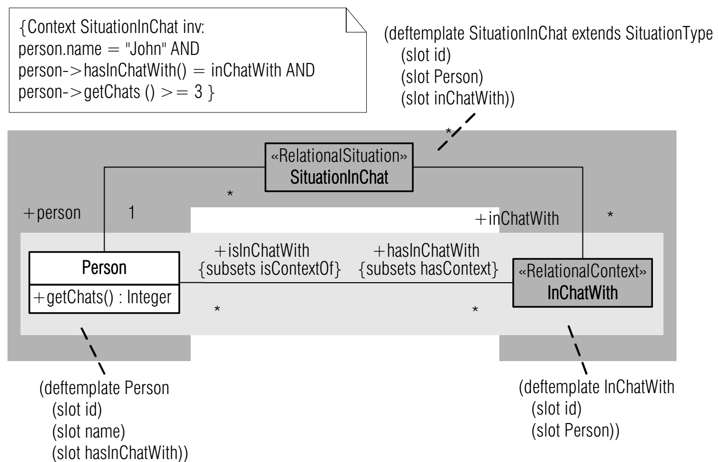
*Figure 6-24* depicts the deftemplates that would be necessary for realizing SituationInChat in DJess. Every object is given a unique identifier, which is used to relate facts. For example, the Person deftemplate has three slots: an identification slot, a name slot and a hasInChatWith slot. The hasInChatWith slot value relates a Person fact to a inChatWith fact. A

person being connected to several inChatWith objects is represented by multiple facts in the working memory. The SituationInChat deftemplate extends the SituationType deftemplate, which defines the initial and final time attributes. In addition, the SituationInChat deftemplate also defines the slots Person and a slot inChatWith, which realize the association ends with these names. Suppose person John is currently involved in three chats, namely a, b and c. This would be represented as follows in the working memory.

```
(Person (id 1) (name John) (hasInChatWith a))
(Person (id 1) (name John) (hasInChatWith b))
(Person (id 1) (name John) (hasInChatWith c))
```

The DJess working memory is similar to a relational database, in which fact templates are similar to tables, facts are similar to tuples, slots are similar to columns, and object unique identifiers are similar to primary keys.

*Figure 6-24* SituationInChat in DJess




The EnterTrue rule for SituationInChat in DJess is similar to the one depicted in *Figure 6-25*, except that method getChats is no longer implemented in class Person, since objects are not being "shadowed" in the working memory. Methods are implemented as static methods, therefore we do not instantiate a class in order to invoke a method. We have implemented getChats method in Util, which is a utility class for implementing static methods.

```
(defrule entertrue_situation_inchatwith
  (Person (id ?id) (name "John"))
  (Person (id ?id) (hasInChatWith ?hasInChatWith))
  (test (>= (call Util getChats ?id) 3))
  (not (SituationInChat (Person ?id) (finaltime nil))
=>
  (bind ?init (call System currentTimeMillis))
  (assert (SituationInChat (Person ?id) (initialtime ?init)))
```

*Figure 6-25* SituationInChat realization in DJess

### 6.5.4 Combining DJess and service-oriented architectures

Yet another realization alternative for the distributed scenario is to use a combination of DJess and service-oriented architectures. In this case, nodes of a DJess network should offer an interface, which provides access to situation events. This combination is only possible because of the flexibility offered by service-oriented architectures, in which components are not required to be implemented in any specific technology. *Figure 6-26* depicts an example of situation detection distribution using a combination of components and DJess. Engines A, B and C participate in a DJess network, in which parts of their working memories are shared in order to distribute knowledge. In addition, engines B and C are encapsulated by components B and C, respectively. These components offer situation event notification services to external requesters, such as component E. Component E subscribes to component B in order to receive notification of occurrences of $situation_2$ events, and subscribes to component C to receive event notification of occurrences of $situation_3$ events.

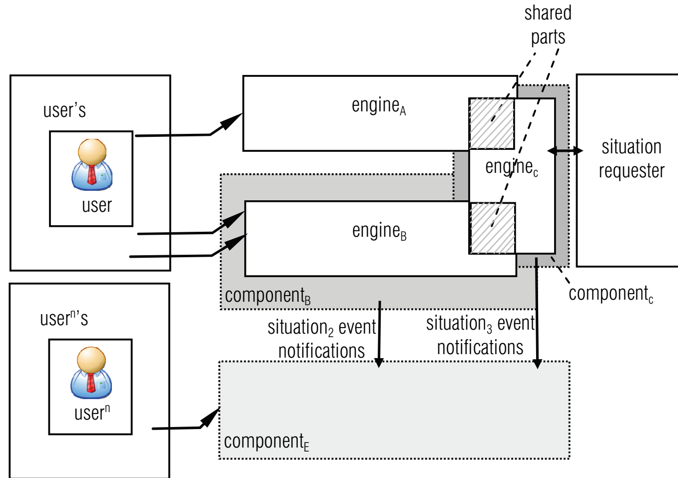



*Figure 6-26* Combination of DJess and service-oriented architecture

## 6.6 Discussion

We have proposed in this chapter a novel model-driven approach for the realization of situation detection for attentive context-aware applications. The situation realization approach is rule-based, and executes on mature and efficient rule engine technology available off-the-shelf. The rule set is derived systematically from the specification and has been deployed directly in the Jess rule engine. Most approaches we have investigated [53, 56, 58, 117] do not provide a model-driven support to the development of context-aware applications. Typically, situations are specified using logic specification languages, such as first order logic or description logics and, contrary to our approach, their realizations are not derived systematically from the specifications. In addition, most initiatives we have studied do not address attentiveness of situation detection. Attentiveness allows situations to be detected simultaneously to their occurrences, enabling applications to be informed of any changes in the state-of-affairs of interest, rather than having to query for specific situations of interest.

We have proposed mappings for basic elements of OCL to the Jess language. Not all OCL expressions are supported by our mapping framework. For example, our mapping approach does not support native OCL expressions to manipulate collections. Because of these mapping limitations, application developers are currently constrained to use the OCL expressions which are supported by our mapping approach (in the realization phase). A more complete mapping framework is necessary, and it is indicated for future work.

We have argued that a distributed solution for situation detection has benefits, which apply in particular to context-aware applications. We have

realized communication between rule engines by means of service-oriented architecture messaging. In this approach, context manager components encapsulate engines, which are capable of communicating with other components through their interfaces.

We have also discussed communication between rule engines with the DJess shared memory mechanism, which allows different engines to execute their rule base on a shared set of facts. Since shadow facts are not supported by DJess, we need to manipulate native Jess facts, in which objects are given unique identifiers, and the working memory is treated similarly to a relational database. Although shadow facts are not supported, there is little impact on how rules are derived in DJess; therefore, in general, our mapping efforts from OCL to Jess still apply for the mappings from OCL to DJess. The benefit of using DJess lies in the complete transparency of situation detection distribution, in which each engine works on the distributed facts as if they were local. However, since shadow facts are not supported, the Java implementation is not so well integrated with the contents of the engine's working memory.

Finally, we have discussed an architectural style in which distribution is realized by combining service-oriented architectures with the DJess middleware. The benefit of using this solution is that it allows application developers to choose particular realization alternatives for different components of the system.

Chapter 7

# Controlling Services

In this chapter we aim at tackling the flexibility, extensibility and adaptability requirements presented in Chapter 2 by means of the *Controlling services*. A controlling service aims at facilitating the dynamic execution of application-specific behaviour specifications using a mobile rule engine and a mechanism that distributes context and situation reasoning activities to context processing components. In Chapter 4 we have discussed that context-aware application-specific behaviours can be described as logic rules, which are called ECA rules, following the Event-Control-Action (ECA) pattern. In this pattern, an Event models an occurrence of interest (e.g., a change in context); Control specifies a condition that must hold prior the execution of the action; and an Action represents the invocation of arbitrary services. In this chapter we discuss the detailed design of the controller component, and we introduce ECA-DL, a domain-specific language used to describe context-aware reactive behaviours following the ECA pattern. In addition, we discuss the relations between the situation detection framework presented in Chapter 6 and ECA rules.

This chapter is further structured as follows: section 7.1 presents the detailed design of the Controller component, which offers controlling services; section 7.2 introduces ECA-DL, its syntax and semantics; section 7.3 discusses how ECA rules can be realized in Jess; section 7.4 elaborates on the integration between controlling services and context provisioning services, and finally section 7.5 presents concluding discussions.

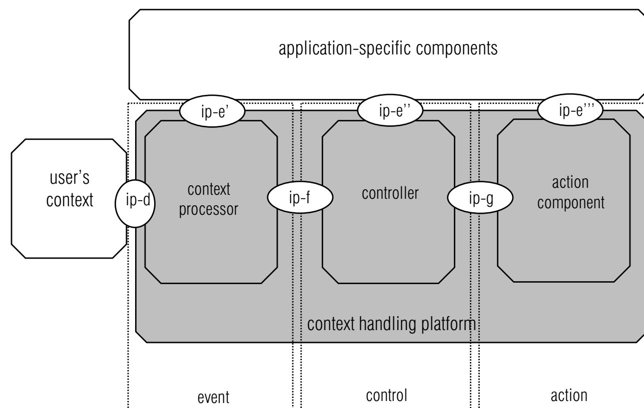
## 7.1 The Controller

The *controller* component aims at tackling flexibility, extensibility and adaptability of the context handling platform, as discussed in Chapters 2 and 4. A controller component receives application-specific behaviour

specifications as input, and executes these behaviours in the platform on behalf of applications. In order to execute application-specific behaviours, a controller component observes events, monitors condition rules, and triggers actions when particular events occur and conditions are satisfied.

*Figure 7-1* depicts the context handling platform as discussed in Chapter 4. The platform is refined into subcomponents, namely *context processor*, *controller*, and *action components*. The use of the ECA pattern (section 4.2) is also represented in this figure by means of dashed rectangles around the components. This figure depicts a logical configuration of context processor, controller and action components. Other configurations are also possible, in which multiple instances of these components interact with each other.

*Figure 7-1* Context handling platform




The interaction points ip-e', ip-e'' and ip-e''' enable application-specific components to interact directly with the different components of the platform. Application-specific behaviours are delegated to the platform through interaction points of type ip-e''. The controller component takes these behaviours as input and configures the rest of the platform to operate properly according to the application-specific requirements. For that, the controller should announce to the context processor components that certain types of context information are needed.

Context processor components are responsible for capturing the user's context through interaction points of type ip-d. Based on measured information of the user's context, context processors generate context information and events, which are observed by controller components through interaction points of type ip-f. Application-specific components may interact directly with the context processor components, through interaction points of type ip-e'. This allows application components to access context information, independently of the availability of the controller component.

When the combination of context conditions defined by the application-specific behaviours is met, the controller component triggers the required actions through interaction points of type ip-g. Application-specific components may also interact directly with action components, through interaction points of type ip-e'''. This allows applications to trigger (combinations of) actions, independently of using the controller component.

### 7.1.1 The controlling service

The controller component offers *controlling service* to requesters. This service allows its users to (i) perform Event-Condition-Action (ECA) rules, and (ii) query for specific instances of context information. As already discussed in section 4.6.4, the controlling service supports the following types of operations: subscribe, unsubscribe, query and notifyApplication. Subscribe activates an ECA rule within the platform; unsubscribe deactivates an ECA rule; query selects specific context information values and notifyApplication notifies application components of the occurrence of ECA events.

ECA rule activation occurs at platform *runtime*, which requires runtime discovery of context provisioning and action services. *Figure 7-1* depicts a typical usage flow of the controlling service. The following phases are identified:

- *Phase 1* initiates with an end-user subscribing to some context-aware application service by means of a graphical interface. This type of service typically requires the application to behave reactively to context and situation changes. The application then delegates the desired reactive behaviours to the platform, which realizes these behaviours on behalf of the application;
- *Phase 2* consists of performing the mapping of an end-user request to a rule specification to be provided to the platform, in a scripting format (e.g., XML). The translation from the user's requests to rule specifications in some notation that can be accepted by the platform is responsibility of the application components. It is also possible that the application developers (as opposed to an end-user) specify application rules at application design-time. End-user and application rules are treated equally here;
- *Phase 3* consists of the actual invocation of the controlling service, in which an application subscribes rule specifications. The controlling service verifies whether the specification is well-formed and separates it into events, conditions and actions;
- *Phase 4* corresponds to the attempt of the controlling service to find context provisioning services capable of providing the context and situation event notifications of interest. The controlling service decides

whether or not to subscribe to one of more of these context provisioning services;

- *Phase 5* consists of exchanging subscribe request messages and possible event notifications, and query requests and answers. The controlling service determines whether the conditions are satisfied by the information provided from context provisioning services;
- *Phase 6* starts typically when a certain condition is satisfied. At this moment, an action should be triggered, and, therefore, its actual implementation needs to be found. For that purpose, the controlling service uses the action discovery service;
- Finally in *phase 7* action service components are invoked by the controller component.

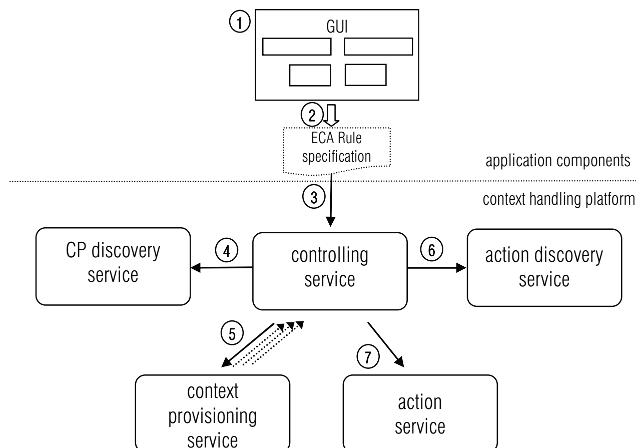



*Figure 7-2* Typical usage flow of the Controlling Service

### 7.1.2 Discovery services

Context sources and managers offer services as in a service-oriented architecture and, therefore, they are registered and discovered in a service repository, as discussed in section 4.6.1. A context source or a manager registers a description of its service (e.g., the type of context or situation information it offers) and the location of an interface where this service is available. Consumers of context information (e.g., the controller and application components) make a query request for a service having certain characteristics, e.g., the service type, costs, location and quality of service parameters. The service registry checks the query request against the service descriptions it holds and responds to the consumer with the location of the selected service's interface. Service registries behave proactively by notifying consumers of newly exported service offers that better match their service query description. We assume context sources, managers and context requesters agree upon the semantics of the context information being exchanged, which has been extensively discussed in Chapter 5.

Upon an ECA rule subscription, the controller component verifies whether context and situation events needed are already available to the controller. If this information is not available, the controller uses the discovery services to find relevant context sources (for context information notifications) and context managers (for situation event notifications), which are capable of offering this information.

The query request may return a set of context sources or managers. We assume all context providers in this response commit to some level of quality, as agreed upon in the query request (see section 5.6.1 for more information on quality). The subscription process with context sources and managers may use one of the following strategies:

- Subscribe to all sources: the controller subscribes to all context sources or managers matched in the query request. In this approach, various notifications are received for the same information. To decide which one to consider, the requester may use quality properties, such as event freshness and accuracy. Although this approach may be beneficial to improve reliability, it is potentially more expensive due to mirror subscriptions and intensive communication traffic (we assume that subscription to services require the payment of fees); and
- Subscribe to one source: the controller selects one context source from the pool of sources resulted in the query request. The selection process can be random or use quality parameters (e.g., the event source that promises the most accurate information).

Since context sources are vulnerable to failure, it may be necessary to perform again a subscription for the same event with a different context source or manager in case of failure. Events re-subscriptions may be necessary in the following situations:

- Quality of the event decreases and it is no longer as required in the service agreement;
- The context source or manager is momentarily unavailable due to unexpected failure; or
- The context source or manager withdraws itself from the platform or goes offline on purpose.

The controller usually detects with little effort whether the quality of event still satisfies the requirements. Whether the source is down either due to failure or withdrawing, is not so straightforward. We assume, from the controller point of view, that there will be management types of event to inform whether a context source is not functioning properly. These events may be generated either by the sources themselves, by a notification middleware platform or some other mechanism.

For time-based events, i.e. events notified in a pre-defined frequency, the detection of whether the event has not been notified at the specified time is straightforward.

### 7.1.3 The Controller detailed design

As already mentioned in the previous section, the controller component should (i) interpret ECA-DL rules, which are provided as input to the controller; (ii) gather context information from various context processor components; (iii) monitor the occurrence of events and conditions; and (iv) invoke actions in response to particular event occurrences and condition evaluations. In order to realize these functions, we have refined the controller component into subcomponents, namely the *ECA handler*, the *event handler*, the *rule engine* and the *action dispatcher*. Each of these subcomponents aims at realizing the aforementioned functions, respectively. *Figure 7-3* refines *Figure 7-1* and focuses on the internal design of the controller component.

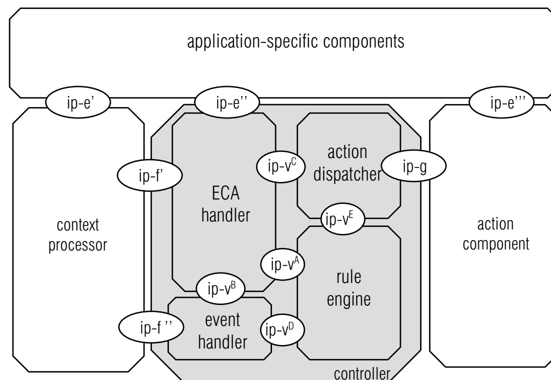



*Figure 7-3* Controller detailed design

The *ECA handler* component accepts ECA rule specifications through interaction points of type ip-e'' and checks whether they are well-formed with respect to the context and situation models shared by the platform components (phase 3 in *Figure 7-2*). Upon the successful validation of the ECA rule specification, the ECA handler sends to the *event handler* component the list of events, context and situation values necessary to evaluate a rule, through interaction points of type ip-$v^B$. Once the rule engine's working memory can be properly populated with context and situation events, the ECA rule specification can be loaded in the rule engine component, through interaction points of type ip-$v^A$.

The ECA handler also verifies whether an action specified in the ECA rule is *critical* (emergency rule). If this is the case, a request for action pre-fetching is forwarded to the action dispatcher through interaction points of type ip-$v^C$, avoiding in this way, action query delays.

Upon receiving information from the ECA handler component, the event handler component verifies whether context and situation events are already available to the controller, and if not, it prepares a query request for each context and situation event that is needed. This query is performed on

a context discovery component, which is not depicted in *Figure 7-3* for the sake of simplicity. The context discovery component returns a set of service offers that fulfil the query requirements (phase 4 in *Figure 7-1*).

The event handler chooses one or more service offers and subscribes to it through interaction points of type ip-f'', in order to receive event notifications (see section 7.1.2). It also performs query-based requests, in which immediate answers with context and situation values are expected. The event handler populates the rule engine's working memory by means of interaction points of type ip-$v^D$. These events are gathered from distributed context processor components through interaction points of type ip-f''. The event handler performs event filtering and pre-processing. In addition, it may detect that a time-based event has not been notified in the expected time, which causes an additional subscription to that event. Throughout this chapter we discuss the various responsibilities of the event handler component in detail.

The *rule engine* component is the core of the controller architecture. It can be implemented on top of a Jess rule engine, which is capable of determining when conditions are satisfied by the current set of available facts in the working memory. In response to a matched condition, an action is triggered through interaction points of type ip-$v^E$.

The *action dispatcher* checks whether it contains a pre-fetched action service for the required action. Otherwise, similarly to the ECA handler, the action dispatcher performs a query request with an action discovery component to find proper action service offers. Finally, the action dispatcher invokes the action component service through interaction points of type ip-g.

## 7.2 ECA-DL

As we have discussed in Chapter 4, the context handling platform should be able to cope with rapid changes in application requirements, and should offer support for reactivity. In order to cope with these aspects, we have developed a domain specific language (ECA-DL) for the purpose of specifying context-aware reactive rules, which are called Event-Control-Action rules (ECA rules). By means of this language, applications developers are capable of specifying rules that express desired context-aware reactive behaviours in a scripting format, which can be deployed at platform runtime. This way, newly defined application requirements that were not anticipated at design time can be realized by activating new rules into the platform. This approach facilitates maintenance of particular application behaviours in the platform, since application developers may add and remove rules on demand without disrupting platform execution.

ECA-DL is defined based on two complementary foundations: information and behaviour. *Information foundation* refers to the representation of the applications' universe of discourse and state-of-affairs, i.e. the context and situation models discussed in Chapter 5. *Behaviour foundation* refers to the dynamics of rule execution, i.e. how and when a rule should be executed and what are the elements of the language that should be used to define a particular piece of reactive behaviour. The following sections elaborate on the details of ECA-DL.

### 7.2.1 Basic concepts

ECA-DL has been developed considering the following requirements:

- Expressive power in order to permit the specification of complex event and condition relations. In order to cope with that, ECA-DL allows the use of relational operator predicates (e.g., $<$, $>$, $=$), and the use of logical connectives (e.g., AND, OR, NOT) to combine events and conditions;
- Convenient use in order to facilitate its utilization by context-aware application developers. ECA-DL provides high-level constructs that facilitate composition of situation events and context conditions;
- Extensibility in order to allow extension of predicates to accommodate events and conditions being defined on demand.

In addition, in order to comply with the ECA pattern discussed in Chapter 4, an ECA rule should consist of the following elements:

- One or more events, which model the occurrence of relevant changes in the user's context. The occurrence of these events triggers the evaluation of the ECA rule;
- Zero or more conditions, which represent the situation under which the actions of the rule are enabled, given that the events have occurred. A condition is typically expressed as a (simple or complex) Boolean expression;
- One or more actions, which represent the operations of the rule that determine the reactive behaviour of the application.

Events, conditions and actions may also have internal structure. For example, a condition may consist of multiple sub-clauses, or an action may be implemented as a procedure call that invokes several sub-procedures.

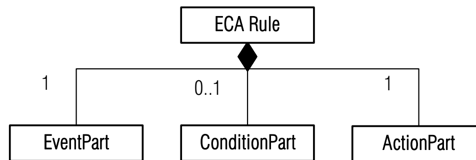
Considering these basic elements that an ECA rule should contain, we have identified the basic requirements for the ECA-DL language elements. ECA-DL should offer means to define the rules containing the following elements:

- An *event* part, which allows defining a relevant situation change by means of simple or complex combination of events;

- An optional *condition* part, which allows defining a logical expression that must hold simultaneously to the occurrence of the specified events and prior to the execution of the action;
- An *action* part, which allows the definition of operation invocations to be executed by the platform as a consequence of the occurrence of the events and the fulfilment of the condition(s) associated with these events.

*Figure 7-4* depict the basic elements of an ECA rule.

*Figure 7-4* Basic elements of an ECA-DL rule




### 7.2.2 Navigation

Navigation aims at reaching values or objects of concern in the context and situation models. Navigating through the context and situation models in ECA-DL is similar to navigating in the OCL language, i.e. in ECA-DL we use dots to navigate from an object to its attributes. In contrast to OCL, in ECA-DL, the target element of a navigation expression is always a primitive datatype, i.e. a numeric, a Boolean or a string value. In addition, we include in the navigation expressions the type of the object being navigated, in order to facilitate the parsing. In general, navigation in ECA-DL starts with an entity, such as EntityType.entityId, where EntityType refers to the type of the entity and entityId refers to the entity's unique identifier. ECA-DL assumes entities have unique identifiers, which are shared by the platform and applications. The elements following the entity's unique identifier aim at navigating through this entity's attributes until the desired attribute's value is reached.

*Figure 7-5* depicts a fragment of a context model. In order to navigate to a device's battery power value, we would define the following in ECA-DL expression: Device.id1.hasBatteryPower.value, where id1 is a unique identifier of a particular entity of type Device. Similarly, in order to navigate to the device's bandwidth value, we would have the expression Device.id1.hasBandwidth.value, and to navigate to the device's current location's latitude value, we would have the expression Device.id1.hasGeoLocation.coordinates.latitude.

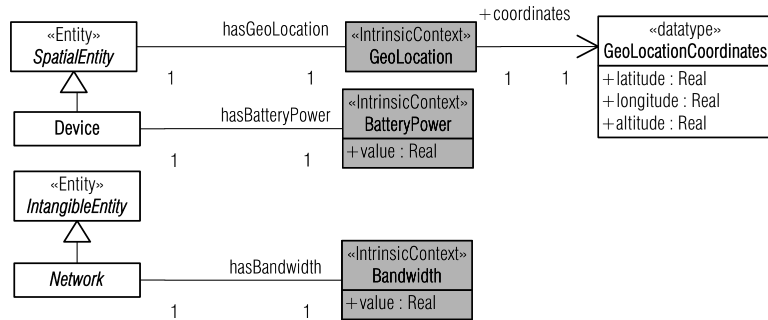



*Figure 7-5* Fragment of context model

Collections in ECA-DL are represented with a star (*) character. For example, the expression Person.* is used to refer to all instances of class Person. Manipulating collections is supported by means of the ECA-DL's Select clause, which is discussed in section 7.2.4.

## 7.2.3 Events

An event represents some happening of interest, which occurs at a specific time [76]. In our approach, an event typically represents changes in the context. Three types of events are supported by our context handling platform, namely *situation events*, *primitive events* and *temporal events*.

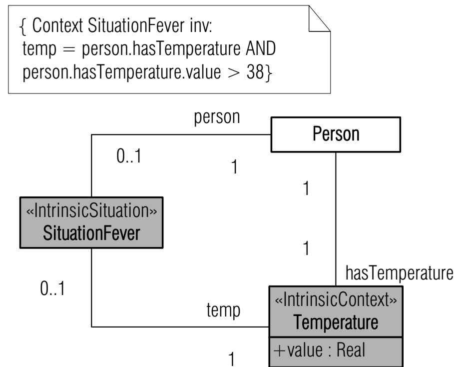
### *Situation events*

*Situation events* are defined in terms of situation transitions, which have been discussed in Chapter 6. Two situation transitions are currently supported, namely *EnterTrue*, and *EnterFalse*. EnterTrue ($S_1$) represents when situation $S_1$ starts to hold, and EnterFalse ($S_1$) represents when situation $S_1$ ceases to hold. Context manager components define interfaces at which it is possible to subscribe to particular transitions in situation state (see section 5.6.2). For example, it is possible to subscribe to a context manager to receive event notifications when a particular situation starts to hold (EnterTrue), or when a particular situation ceases to holds (EnterFalse).

*Figure 7-6* depicts a simple example of situation specification, namely SituationFever, which defines that a person's temperature is above 38 degrees Celsius. If we would like to refer to the event in which John starts to have fever, we would define the following in ECA-DL: EnterTrue (SituationFever (Entity.John))[4]. Similarly, if we would like to refer to the event that occurs when John no longer has fever we would have the following expression in ECA-DL: EnterFalse (SituationFever (Entity.John)).

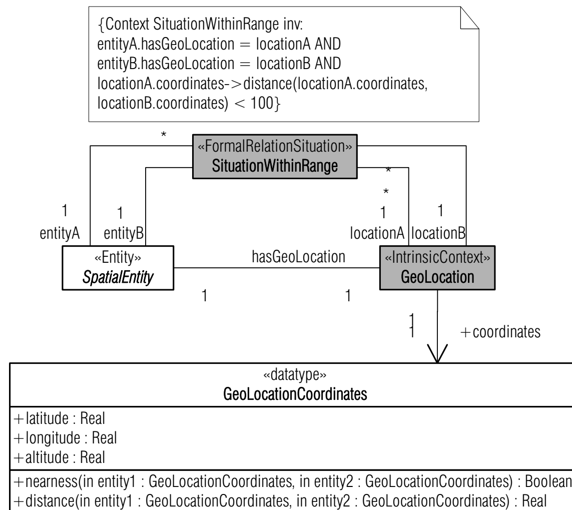
[4] We often use in our examples the person's name as the person's unique identifier to improve readability of the examples. Although this is not a problem in our examples, in real applications, proper unique identifiers should be considered.

*Figure* 7-6 SituationFever specification




In general, in order to verify whether a specific entity is currently in a particular situation, we provide the entity's unique identifier as a parameter, such as SituationType (EntityType.entityId). If the situation specification involves more than one entity, these entities can be provided as complimentary parameters. Parameters serve as filtering expressions to refer to situations that are more specialized than the original situation specification. For example, SituationType (EntityType.entityId) is more specialized than SituationType (), and SituationType (EntityType.entityId, EntityType.entityId2) is more specialized than SituationType (EntityType.entityId). *Figure* 7-7 depicts a situation specification example, in which the location coordinates between two special entities are compared in order to check whether these entities are within 100 meters distance from each other.

*Figure* 7-7 SituationWithinRange example




The ECA-DL expression SituationWithinRange() refers to all instances of this situation regardless the spatial entities which are within range. Differently,

the expression SituationWithinRange(Person.John) regards all instances of this situation in which John is involved, independently of the second entity that is within 100 meters from John. Expression SituationWithinRange(Person.John, Device.PDA) regards the situation in which John is within 100 meters from his PDA. When subscribing to notifications of less specialized situation events, such as, for example, EnterTrue (SituationWithinRange ())), potentially many event notifications are generated, each event notification reporting a pair of entities that gets within 100 meters from each other. The number of event notification potentially decreases when subscribing for more specialized situations. For example, subscribing to EnterTrue (SituationWithinRange(Person.John, Device.PDA)) generates event notifications only when John and his PDA get closer than 100 meters.

The situation SituationWithinRange specifies a *symmetric* relation between two spatial entities, i.e. the order of the entities participating in this situation is irrelevant. For example, John being close to his PDA means the same as John's PDA being close to John. Other situations may specify *asymmetric* relations between entities, such as the situation SituationContained (container, contained), which specifies a containment relationship between two entities, the *container* and the *contained* entity. In such situations, the filtering expressions must respect the ordering of the attributes, since SituationContained (Person.John, Building.HouseJohn) is not the same as SituationContained (Building.HouseJohn, Person.John). The order of the parameters of a situation specification is defined respecting the order to the attributes in the respective situation measurement datatype, which is part of the situation information models. We have already discussed these issues in Chapter 5.

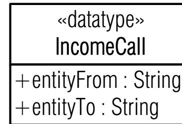
With respect to the filtering expressions, the parameters that are not intended to be matched are left *blank*. For example, if we would like to know when anyone enter John's house, we would use the filtering expression EnterTrue (SituationContained (Building.HouseJohn, )). Similarly, if we would like to know when John leaves any container entity, we would use the filtering expression EnterFalse (SituationContained ( ,Person.John)).

### *Primitive events*

*Primitive events* are the events that cannot be detected by means of situations. As opposed to situation events, primitive events are generated by components that do not necessarily implement our situation detection mechanism. An example of primitive event could be IncomeCall (entityFrom, entityTo), which represents the moment there is an incoming call from entity entityFrom to entity entityTo. As discussed in Chapter 5, primitive events are defined as datatypes in the context information modelling phase, and the possible parameters of a primitive event are defined as attributes of the

primitive event datatype. *Figure 7-8* depicts the specification of the IncomeCall primitive event.

*Figure 7-8* IncomeCall primitive event datatype specification




Filtering primitive events is similar to filtering situation events. For example, expression IncomeCall (,Person.John), regards all IncomeCall event notifications in which John is the callee. Similarly, expression IncomeCall (Person.John, ) regards all IncomeCall event notifications in which John is the caller. Finally, the expression IncomeCall (Person.Mary, Person.John) regards all the IncomeCall event notification in which Mary is the caller and John is the callee.

***Temporal events***

*Temporal events* regard the events that are generated from time to time, at some specific frequency. These types of events aim at notifying that a certain period of time has passed. For example, an application would like to receive an event notification every 30 minutes. The happing of interest being notified in this example is that, since the last notification, 30 minutes have passed. An example of ECA-DL expression that refers to temporal events is OnEvery(t), which specifies that there is a temporal event generated every t milliseconds. Temporal events can be generated and handled within the controller component, with no need to perform event subscriptions on external components.

***Event composition***

Context-aware applications often require the specification of behaviours that refer to complex event compositions, rather than simple events. For example, a particular piece of application behaviour may require the specification of the event "John enters the room and at most 5 minutes later he leaves the room", or "John approaches Mary, they stay nearby each other for at most 20 minutes, and no one else approaches John when John and Mary stay nearby". In order to support such types of event compositions, we have considered the approach implemented by the event distribution framework presented in [76].

In our approach, event composition expressions can combine primitive and situation events using the event operators depicted in *Table 7-3*. The result of an event composition is called a *composite event*.

*Table 7-3* Event composition operators

| Operator | Composite event |
|---|---|
| e1&e2 | Occurs when both e1 and e2 occur irrespective of their order |
| {e1;e2} ! e3 | Occurs when e1 occurs followed by e2 and e3 does not occur between them |
| e1 \| e2 | Occurs when e1 or e2 occurs |
| e1;e2 | Occurs when e1 occurs before e2 |

As an example, consider the operator ";" to specify the behaviour "John enters the room and at most five minutes later he leaves the room". The event composition specification would be as follows:

EnterTrue (SituationContained (Room.3040, Person.John)) ; EnterFalse (SituationContained (Room.3040, Person.John))

Still, this specification does not express the correct behaviour, which says that John should leave the room at most 5 minutes after he enters. This is because the operators shown in *Table 7-3* alone do not provide means to specify particular temporal constrains besides precedence. It is often necessary to specify particular temporal aspects of application's behaviours. In order to (partially) overcome this, we define the *occurrence interval of a* composite event. An *occurrence interval* represents the time period during which the composite event is being detected, since composite events are detected in terms of more elementary events (primitive or situation events). An elementary event occurs at a specific time, which is called the *occurrence time*.

The first elementary event of an event composition, which starts the occurrence interval at time $t_1$, is called the *initiator*. The last one, which causes the composite event to occur at time $t_n$, is called the *terminator*. We define that the actual occurrence time of the composite event is $t_n$, i.e. the terminator's occurrence time.

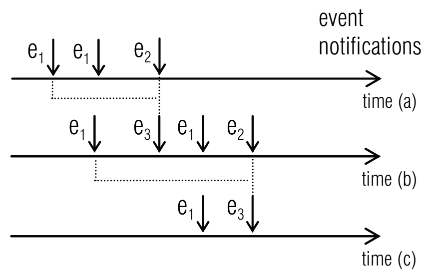
Another aspect of event composition is that elementary event notifications cannot be considered indefinitely for detecting a particular composite event. Suppose we would like to detect the event "John enters the room and at most 15 minutes later he turns his computer on". Consider that before turning his computer on, John has entered the room 3 times in the past hour, more precisely, at 15:05 hrs, 15:10 hrs, and at 15:13 hrs, respectively. When John finally turns his computer on at 15:15 hrs (terminator), the composite event should be detected, and the initiator event should have occurred at most 15 minutes before the terminator. In this example, the past three times John has entered the room have occurred at most 15 minutes before he turned his computer on. We define in our approach that the oldest occurrence of a particular event should be considered as the initiator. Therefore, the notification generated at 15:05

hrs when John has entered the room, is considered the initiator of this particular event composition.

Suppose now that John turns his computer on again at 15:18 hrs, which should generate another composite event. Again, three possible occurrences are within the occurrence interval. In order to avoid that the same elementary event occurrence is considered in different event composition detections, we define that particular elementary event notifications should be considered only *once* in the composition. Therefore, we should not consider the 15:05 hrs occurrence again, since it has been already considered for composition. Since event notifications are considered only once per event composition detection, we say that the event is *consumed* by the composition detection.

Since an ECA rule is responsible for detecting composite events, an ECA rule is said to *consume* primitive events in order to detect the occurrence of a composite event. For example, consider the composite event e3: (e1&e2), which should be defined in the scope of an ECA rule R1. Consider that the following sequence of event notifications would be received by the controller component: e1, e1, e2, e1, e2. The composite event e3 is detected twice, as depicted in *Figure 7-9*. The first time event e3 is detected (timeline (b)), both events e1 and e2 are consumed by R1, i.e. they are not considered again for detection of e3. In the second time e3 is detected (timeline (c)), the oldest occurrence of e1 and e2 are consumed by R1, and, therefore, not considered again for event composition detection.

*Figure 7-9* composite event detection




Suppose now that before turning his computer on, John has entered the room 3 times in the past hour, more precisely, at 15:00 hrs, 15:40 hrs, and at 15:45 hrs, respectively. When John finally turns his computer on at 15:48, the composite event should be detected, and the initiator event should have occurred at most 15 minutes before the terminator. Therefore, the event notification generated when John entered the room at 15:00 hrs should be discarded, since it occurred 48 minutes before he turned on this computer, falling outside the maximum period of time determined by the application behaviour. In order to cope with elimination of events that fall

outside a particular period of time, we defined the *detection window interval* concept.

We say that a particular composite event is detected inside a *detection window interval*, which defines the *maximum* occurrence interval for an event composition. This means that the time between the terminator and the initiator events should be never greater than the detection window interval. Event occurrences outside the detection window are not considered for the composite event detection. Since different application behaviours require different detection window interval values, a particular detection window interval is defined per ECA rule, at rule subscription time. A default value is set by the controller component if no value is provided by the application developer. *Figure 7-10* depicts the detection of composite event e3, considering a detection window interval *dw*. When the notification of event e2 is received, e3 is detected because of the second occurrence of e1 and not the first occurrence. Since the first occurrence of e1 falls outside the detection window interval, it is not considered for the detection of e3.

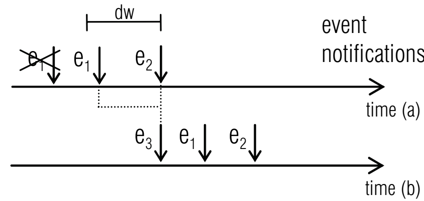



*Figure 7-10* Small detection window interval in a composite event detection

*Figure 7-11* depicts an example of a slightly larger detection window interval. Since the detection window now is larger than before, the first occurrence of e1 is considered for detecting e3. However, in timeline (b), the second occurrence of e1 is discarded, since it falls out of the detection window. In this case, event 3 is detected with the third occurrence of e1.

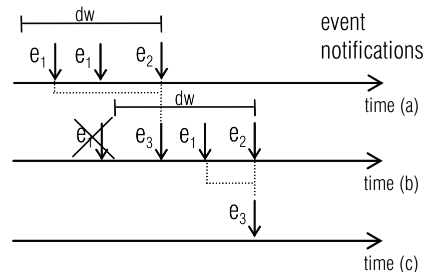



*Figure 7-11* Larger detection window interval in a composite event detection

## 7.2.4 Syntax and Semantics

In the following sections we introduce the ECA-DL's concrete syntax and semantics. For each of the supported clauses, we present (i) the behaviour intended with this clause (semantics); and (ii) how the clause can be used

(syntax). In addition, we present the ECA-DL's metamodel, which graphically represents the elements of ECA-DL, and shows how these elements relate to each other.

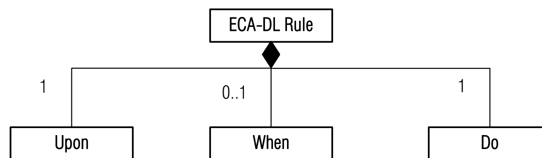
***Upon-When-Action clauses***

We have presented in section 7.2.1 the requirements for the ECA-DL regarding the need to represent events, conditions and actions in a rule. These requirements resulted in the ECA-DL clauses Upon, When and Do, respectively. Events are defined in the Upon clause, while conditions are specified in the When clause and, finally, actions are specified in the Do clause. The When clause may be omitted if there are no conditions to be specified. This way, an ECA-DL rule has the following main structure:

```
Upon <uponExpression>
When <conditionExpression>
Do <actionExpression>
```

In the specification shown here, each of the ECA-DL rule parts are indicated in the syntax on a separate line by *tokens* (a "primitive block of structured text", usually a single word) and their arguments. Syntax is denoted here using a different font; non-literal tokens are placed between angled brackets. *Figure 7-12* depicts a UML diagram which represents a fragment of the ECA-DL metamodel.

*Figure 7-12* Fragment of ECA-DL metamodel focusing on top elements



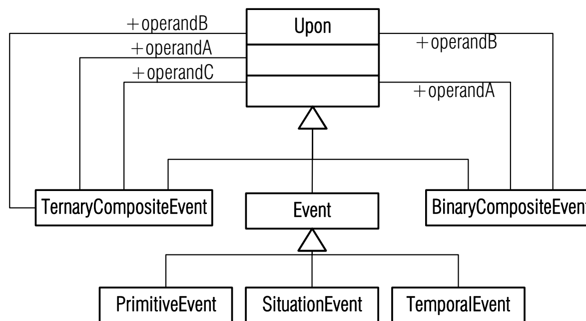

An <uponExpression> can be (a combination of) situation events and primitive events, or it can be a temporal event, which is used individually. These events have been discussed in section 7.2.3. The occurrence of the (composite) event specified in the Upon clause triggers the evaluation of the When clause. Events can be combined using the operators depicted in *Table 7-3*. In addition, it is possible to assign variables to specific events, so that meta-information about these events, such as the event occurrence time and parameters, can be evaluated in other parts of the rule.

*Figure 7-13* depicts a fragment of the ECA-DL metamodel focusing on the possible elements of an Upon clause, which are Event, TernaryCompositeEvent and BinaryCompositeEvent. An Event can be a situation event, a primitive event or a temporal event. A situation event (SituationEvent) is composed of a transition (EnterTrue or EnterFalse), a situation type, and possibly parameters, which are always entities. A primitive event (PrimitiveEvent) is composed of an

event description and possibly parameters, which can be entities, context values, attribute values, literals and function invocations. Details about these types are discussed in the next paragraphs. A temporal event (TemporalEvent) is currently supported by the function OnEvery (t), as explained in the section 7.2.3. *Figure 7-18* depicts the ECA-DL's complete metamodel.

The BinaryCompositeEvent type consists of a composite event, in which the composition is specified in terms of the binary operators &, | and ;. The TernaryCompositeEvent type consists of a composite event, in which the composition is specified in terms of the ternary operator {;}!.

*Figure 7-13* Fragment of ECA-DL metamodel focusing on the UponExpression




A < actionExpression > consists of a list of function invocations. A function is a type of Term, which is explained in the following paragraphs.

A <conditionExpression> is a Boolean expression that can be composed of other Boolean expressions (recursively) by means of binary and unary Boolean operators. The binary operators (and, or) are composed of two operands, namely operandA and operandB. The unary operator not is composed of one operand, namely operand. The inner most occurrences of these operands are also Boolean expressions, which can be:

- Navigation expressions (section 7.2.2) that refer to a Boolean attribute. For example, the navigation expression Person.John.hasMsnStatus.value refers to a Boolean value;
- A function invocation, in which the return of the function is a Boolean value. Function invocations are explained in the next section;
- A binary comparison expression using the comparison operators > (greater than), < (less greater than), = (equal), <> (different). Each operand of these types of binary expressions should be resolved to a numeric or a string value. An example of comparison between navigation expressions is Person.John.age > Person.Alice.age, and between a navigation expression and a literal is Person.John.age>40;

*Figure 7-14* depicts a fragment of the ECA-DL metamodel focusing on the possible elements of a When clause.

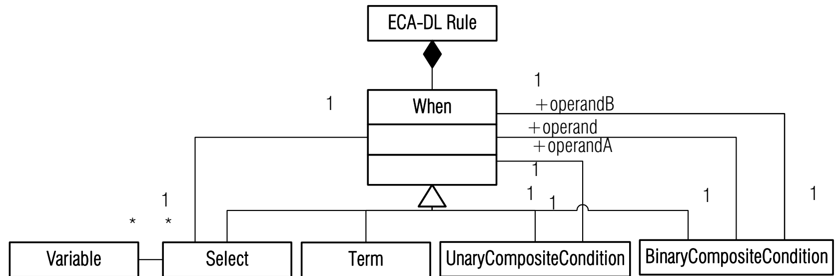



*Figure 7-14* Fragment of ECA-DL metamodel focusing on possible elements of a When clause

A When clause can be one of the following elements:

- A unary composite condition (represented by class UnaryCompositeCondition), which is composed of the Boolean operator not followed by its unique operand (not <expression>). This <expression> is represented in the model by an association end operand between classes UnaryCompositeCOndition and When. An operand can be any of the When elements;
- A binary composite condition (represented by class BinaryCompositeCondition), which may be composed of one of the following operators (and, or, >, <, =, <>), and its two operands. These operands may be again any When element, and are represented by the association ends operandA and operandB between classes BinaryCompositeCondition and When;
- A Select (represented by class Select), which allows selecting a collection of entities respecting a particular filtering expression. A select expression is defined in detail in the following sections;
- A Term (represented by class Term), which can be a context situation (ContextSituation), an entity argument (EntityArgument), an entity's context value (EntityContext), a collection of entities (Entities), an entity's attribute value (EntityAttribute), an event parameter (EventParameter), a literal (Literal) or a function (Function). *Figure 7-15* depicts a fragment of the ECA-DL metamodel focusing on Term.

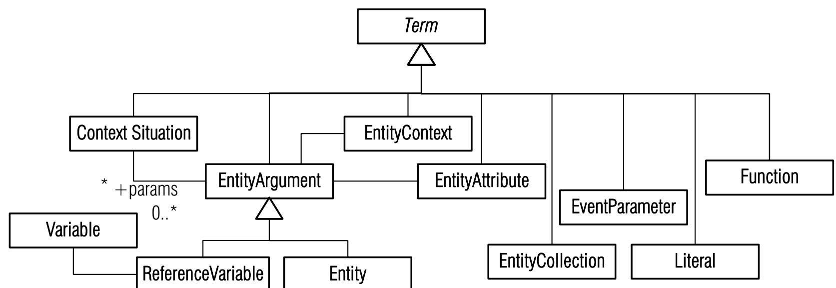



*Figure 7-16* Fragment of the ECA-DL metamodel focusing on a Term

An entity argument (EntityArgument class) is an element that refers to a specific entity, by means of an *explicit entity* (Entity class) or by means of a *reference*

*variable* (ReferenceVariable class). An explicit entity refers to the element with the format EntityType.entityid, for example, Person.id1, where id1 is a unique identifier for that particular person. A reference variable entity uses a variable to refer to an entity. For example, the expression variable.attribute uses the reference variable variable to refer to a particular entity. Variables are defined with Scope and Select clauses, which are discussed in the next sections.

A context situation (ContextSituation) may be associated with a set of entity argument parameters, which are represented by association end param between classes ContextSituation and EntityArgument. Examples of situation context expressions are SituationContained (Person.id1), SituationContained (variable, Person.id1), and SituationWithinRange (variable1, variable2).

An entity's context (EntityContext) refers to a context information value associated with an entity. Therefore, an entity's context expression is composed of an entity argument expression, and a sequence of possible association attributes, such as EntityType.entityid.[ContextType.attribute], in which expressions of type [ContextType.attribute] can be repeated, if the context model allows that. For example, considering the context model depicted in *Figure 7-5*, the following EntityContext expression is valid: Person.id1.hasGeoLocation.coordinates.latitude. Similarly, a variable may substitute the expression EntityType.entityid, for example, and the expression variable.hasGeoLocation.coordinates.latitude is also valid.

An entity's attribute (EntityAttribute) refers to an attribute associated with that particular entity, such as, for example, the age and address of a person. An entity's attribute expression is composed of an entity argument expression, and one possible attribute, such as EntityType.entityid.attribute. For example, the following expressions are valid: Person.id1.age, Person.id1.height, variable.age.

A collection of entities (EntityCollection) refers to a collection of entities of a particular type. An collection of entities expression follows the format EntityType.*. An example of expression that refers to the collection of all persons is Person.*.

An event parameter (EventParameter) refers to a specific parameter of an event that has been specified in the Upon clause, for which a variable is assigned. For example, suppose we define in the Upon clause the filtering event expression E2:IncomeCall(Person.id1,). In the When clause, we can refer to the parameters of that particular event notification, such as E2.entityTo, which refers to the callee of that particular incoming call. In general, an event parameter expression has the format eventVariable.eventParameter.

A function (Function) refers to a function invocation, which can accept a list of arguments. For example, the expression SendSms (Person.id1), refers to the invocation of function SendSms, passing entity Person.id1 as argument.

***Functions***

Often ECA-DL expressions refer to *functions,* which represent invocations to operation either defined internally or externally to the platform. Three types of functions are supported in ECA-DL:

– *Auxiliary functions* refer to the operations which are used as helper functions, such as, for example, the function count (collection), which returns the number of elements of a collection. Auxiliary functions are defined and resolved within the controller component;
– *Formal relation functions* refer to the formal relation operations defined in the datatypes of our context and situation models (see *Figure* 7-7). Examples of formal relations functions are distance and nearness;
– *Action functions* consist of invocations to both internal and external services (with respect to the platform). Typically an action consists of external service invocations, such as a request for an SMS delivery (sendSMS (Person.John)) or starting a telephone call between two persons (TelephoneCall (Person.John, Person.Mary)).

***Select clause***

The Select clause allows filtering collections respecting particular conditions. With the select clause we can retrieve a subset of a collection respecting a variety of constraints. For example, it may be necessary to select all persons that are in a house, or we would like to select all devices that are currently being used, or even all the patients of a clinic who have diabetes. The Select clause syntax is as follows:

```
Select (<collection-of-entities>; <var>; <filtering-expression-involving-var>)
```

The <collection-of-entities> expression contains the original collection, for example, Person.*, which refers to all instances of class Person; the <var> expression aims at naming each element of this collection with a string literal; and the <filtering-expression-involving-var> defines a condition involving <var> which should hold for all elements of the resulting collection.

As an example of select clause, consider the expression Select (Person.*, p, p.age>40), which returns a collection with all the persons whose age is above 40 years old. Another example is (Person.*, p, SituationWithinRange (p)), which returns all the persons currently in SituationWithinRange.

Select clauses may be nested. The <collection-of-entities> expression may be the result of a select evaluation. Variables defined in one occurrence of the select clause cannot be used in another occurrence. Nesting select clauses has a similar effect of including extra constraints in the filtering expression. The following expression exemplifies nesting of select clauses: Select (Select (Person.*, p1, p1.age>40), p2, SituationWithinRange (p2)), which selects all persons above 40, and who are currently in SituationWithinRange situation.

***Scope clause***

ECA rules can be either parameterized or not. Parameterization is necessary when the rule should be applied to a collection of entities. It would be cumbersome to write a rule for each target entity. Instead we define a scope, which specifies a collection of entities for which a single ECA rule should be applied. For example, a medical clinic would like to apply a general rule (notify when sugar levels go above 110) to all patients suffering from diabetes. Parameterization allows the specification of a single rule to be executed for all the diabetic patients. We have introduced the Scope clause to define rule parameterization, and its syntax is:

```
Scope (<collection-of-entities>; var)
{ Upon < eventExpression >
  When <conditionExpression>
  Do <actionExpression>
}
```

The <collection-of-entities> expression defines the collection of entities for which the following ECA-DL rule should be applied. This collection of entities may be the result of a filtering expression using the Select clause. The variable var is a reference to each element in <collection-of-entities>.

For example, the following scope clause defines that for each person (p), who is currently located in his/her own house (SituationContained (person, person.house)), the rule that follows should be applied. This rule defines that upon the entrance of any person in the house (EnterTrue (SituationContained ( p.house))), person p should be notified with a message (Notify (p, "there is someone entering the house")).

```
Scope (Select (Person.*, person, SituationContained (person, person.house)), p))
{
  Upon EnterTrue (SituationContained ( p.house))
  Do Notify (p, "there is someone entering the house")
}
```

Figure 7-17 depicts a fragment of the ECA-DL metamodel focusing on the When and Do clauses. We have included in this figure the scope and select clauses. A scope clause defines a variable, and refers to a collection of entities, which can be of type EntityCollection, or a Select. A Select clause refers to a collection of entities (of type Entities), and defines a variable representing each element of this collection. In addition, a select clause is associated to either a UnaryCompositeCondition or a BinaryCompositeCondition, which aims at defining conditions to filter a particular collection.

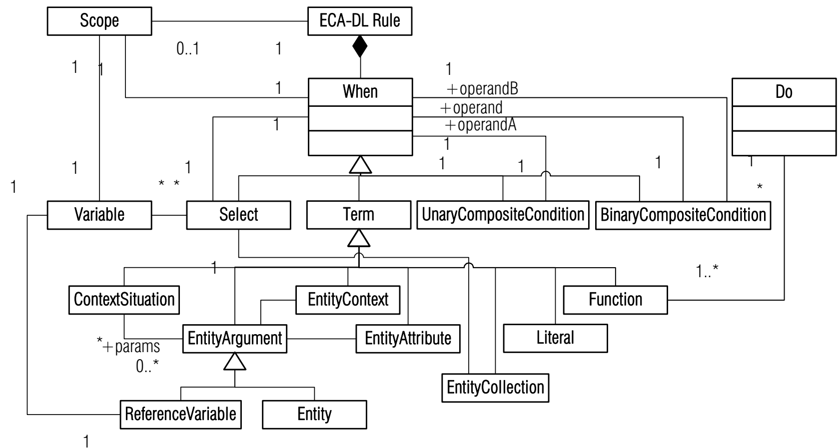



*Figure 7-17* Fragment of the ECA-DL metamodel focusing on When and Do elements

### *Lifetime*

In ECA-DL, an additional statement can be used to indicate the lifetime of the rule. The following lifetime statements are supported:

- always activates the rule and keeps it active. This is the default;
- once activates the rule and deactivates the rule after it has been executed once;
- <n> times activates the rule and deactivates it after it has been executed <n> times;
- from <start> to <end> activates the rule at <start> and deactivates it at <end>. <start> and <end> are moments in time, such as, e.g., "May 1st, 2007";
- to <end> activates the rule, and deactivates the rule at <end>;
- frequency <n> times per <period> activates the rule, and keep it active as long as it has been executed fewer than <n> during <period>. It deactivates the rule and keeps it inactive as long as it has been executed <n> times during <period>.

For example, the following ECA rule notifies Mary that John has entered her house. Since the lifetime is once, the rule is executed only once and deactivated.

```
Upon EnterTrue (SituationContained (Person.John, Person.Mary.house))
Do Notify (Person.Mary, "Mary, John has entered the house")
once
```

### *ECA-DL Metamodel*

*Figure 7-18* depicts the complete metamodel of the ECA-DL language. *Figure 7-18* shows the associations among Upon and When elements. Primitive events may have arguments, and each of these arguments is a Term. These arguments are represented by the association between

PrimitiveEvent and Term classes. A situation event consists of an event transition (EnterTrue or EnterFalse) and a context situation. This relation is represented by the association between SituationEvent and ContextSituation classes.

*Figure 7-18* ECA-DL metamodel

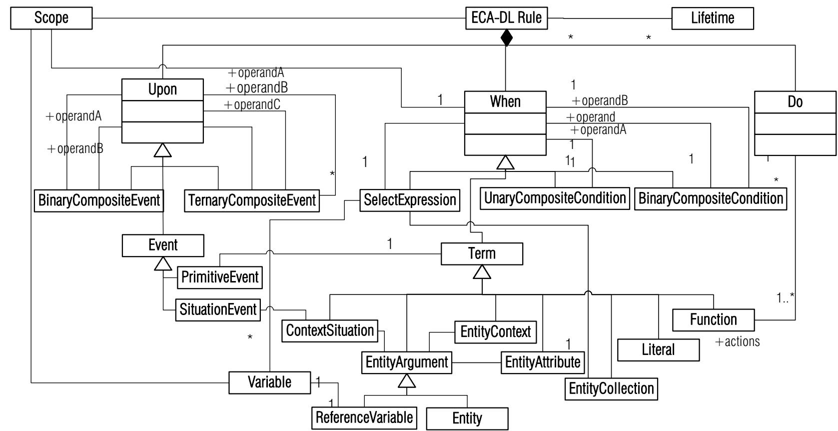



*Table A-4* summarizes the main clauses of ECA-DL.

*Table 7-4* ECA-DL clauses

| ECA-DL clause | Syntax |
|---|---|
| Upon-When-Do | Allows specification of an action (Do) which is triggered upon the occurrence of an event (Upon) respecting general conditions defined in the When clause. Syntax:<br>Upon < eventExpression ><br>When <conditionExpression><br>Do <actionExpression> |
| Select | Allows the selection from a collection using filtering expressions (logical expressions). Syntax:<br>Select (<collection-of-entities>; <var>; <filtering-expression-involving-var>) |
| Scope | Allows parameterization of ECA rules. Syntax:<br>Scope (<collection-of-entities>; var)<br>{ Upon < eventExpression ><br>When <conditionExpression><br>Do <actionExpression><br>} |

### 7.2.5 ECA rule execution

There may be various execution alternatives for realizing ECA-DL rules in a rule engine, respecting the semantics of the language as discussed in

previous chapters. For example, we may check the conditions before expecting events, or the other way around. Here we discuss a possible execution alternative, which is used in our realization approach.

In general, the execution cycle of an ECA rule consists of detecting events, checking conditions and invoking actions. Upon the activation of an ECA rule within the controller component, the rule engine starts to continuously detect whether the events defined in the Upon clause have occurred or not. *Figure 7-19* depicts the execution cycle for the lifetime always, which is the default value. Continuously detecting events take place in the DetectEvent state. When the (combination of) events occur, the engine checks whether the condition defined in the When clause is either true or false. Checking the condition takes place in the CheckCondition state. If the condition is evaluated to true, the actions are invoked, which takes place in InvokeActions state. Regardless of whether conditions are evaluated to true or false, after leaving the state CheckCondition, the rule engine starts detecting events again. This cycle continues until the ECA rule is removed from the controller component.

*Figure 7-19* ECA rule execution cycle for the lifetime *always*

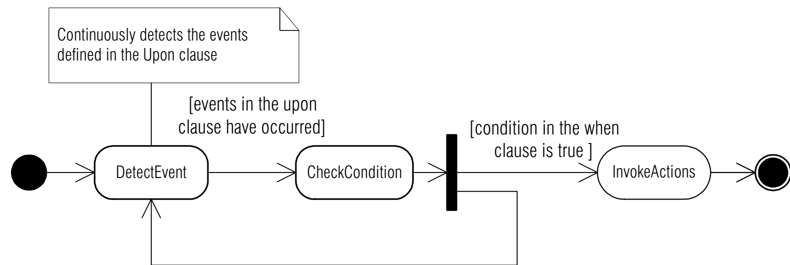



*Figure 7-20* depicts the execution cycle of an ECA-DL rule for the lifetime *frequency* <*n*> *times per* <*period*>. In this case, it is necessary to keep information about the n-th past action execution. If the n-th execution time is outside <*period*>, there is still room for another action execution within <*period*>. Therefore, the action is executed, and information about this execution, such as execution time, is stored. If the n-th execution time is inside <*period*>, n executions have already occurred within <*period*>. Therefore, the action should not be invoked. The execution cycle of an ECA rule for the other possible lifetime constrains are discussed in Appendix B.

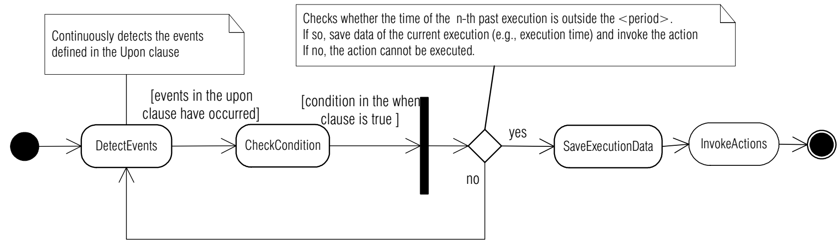



*Figure 7-20* ECA rule execution cycle for the lifetime frequency *<n> times per <period>*

### 7.2.6 Examples of ECA-DL rules

Below we give some examples to illustrate the use of the ECA-DL in different application domains. The following ECA-DL rule specifies that whenever there is an epileptic alarm event for user John, and he is currently driving, he should receive a warning SMS.

```
Upon EpilepticAlarm (Patient.John)
When SituationDriving (Patient.John)
Do SendSms (Patient.John, "John, you may have an epileptic seizure, please stop the car")
```

The following ECA-DL applies a scope clause to constrain the epileptic alarm rule for the patients currently located in Enschede.

```
Scope (select (Patient.*; patient; patient.type = "epileptic" and patient.hasCivilLocation.city
= "Enschede") ; p)
{
    Upon EpilepticAlarm (p)
    When SituationDriving (p)
    Do SendSms (p, "You may have an epileptic seizure, please stop the car")
}
```

The following ECA-DL rule sends a warning to diabetic persons in case of a high sugar level alarm.

```
Scope (select (Patient.*; patient; patient.type = "diabetic") ; p)
{
    Upon HighSugarAlarm (p)
    Do SendSms (p, "You have high sugar levels")
}
```

The following ECA-DL rule applies a scope clause to constrain the persons entering a cinema (Cinestar) in Enschede, so that each person receives an SMS with an advertisement.

```
Scope (Select (Person.*; person; person.hasCivilLocation.city = "Enschede"); p)
{
    Upon EnterTrue (SituationContained (p, Building.Cinestart))
    Do SendSms (p, "Welcome to Cinestar, we have special deals today!")
}
```

The following ECA-DL rule allows policemen working in the field to see his/her colleagues that are within 100 meters from each other. It uses a temporal event that allows the rule to be executed every 5 seconds. The auxiliary function List returns a collection in which the head is one policemen, and the rest of the collection contains the other policemen that are within 100 meters from the first. SituationWithinRange is used to verify whether the policemen are within 100 meters from one another.

```
Scope (Select (Policeman.*; policeman; policemen.hasActivity.value = 'working'; p1)
{ Upon OnEvery (5)
  Do NotifyApp (application-address,
    List (p1.id, select (policeman.*, p2; SituationWithinRange (p1, p2) and
            p2.hasActivity.value = 'working')))
```

The following ECA-DL rule sends a warning to John when he leaves the house and forgets the lunch box inside the house.

```
Upon EnterFalse (SituationContained (Person.John, Building.HouseJohn))
When SituationContained (LunchBox.id, Building.HouseJohn)
Do SendSms (Person.John, "John, you forgot your lunchbox!")
```

## 7.3 Realization of ECA-DL Rules in Jess

There may be various realization alternatives for implementing ECA-DL. We have attempted to implement an ECA-DL compiler [33], which did not succeed in providing us with the required levels of performance and scalability. In order to cope with these aspects, we have decided to use a mature rule-based technology available off-the-shelf to realize ECA-DL.

As discussed in Chapter 6, we have chosen Jess [42] for realizing our rule-based situation approach. Similarly, we also use Jess to realize ECA-DL rules. ECA-DL is a domain specific language, i.e. it has been designed with the purpose of defining context-aware reactive behaviours. The Jess language, on the contrary, is a general purpose language, suitable for a diversity of domains. For this reason, writing context-aware behaviours in ECA-DL requires less programming effort than in the Jess language. It is often the case that realizing a single ECA-DL rule in Jess requires several

Jess rules. Since ECA-DL rules are not directly supported by Jess, we discuss possible mapping alternatives from ECA-DL constructs and expressions to Jess constructs and expressions. In Chapter 6 we have discussed the details of the Jess architecture and the Jess language.

In addition, we assume for the mapping that the working memory in the controller's engine contains the necessary information to execute the rules at any moment in time. The content of the working memory is acquired from context sources, context managers, and local sources. Information from context sources and managers is typically remotely acquired by, for example, subscribing to information provided by these components. Local information is provided to the controller by the platform developer (at platform design time) or the platform provider (at platform runtime).

We also assume that our context and situation models, which are necessary to understand ECA-DL rules, have been implemented in the Java language, as we have discussed in section 5.4. Similar to the situation realization approach discussed in Chapter 6, the controller also uses the shadow fact mechanism to integrate the model Java implementation with the working memory. In this way, modifications to the Java objects are automatically synchronized with the working memory.

### 7.3.1 General mappings

The general structure of a Jess rule has the form `if <conditions> then <actions>`. For the rule to be applicable, i.e. to invoke the actions, or to derive the conclusions, the conditions should be true. An essential difference between ECA-DL and Jess is that events are not directly supported by Jess. In ECA-DL, the event part (upon clause) describes when the rule should be executed and contains event expressions (section 7.2.3). In addition, a possible condition part (when clause) specifies additional constraints that should hold before the actions are invoked. Therefore, the Upon clause of an ECA-DL rule cannot be simply mapped to a Jess condition.

The general mapping mechanism we use is the following: for each ECA-DL rule, at least two Jess rules are generated, namely one that consumes the event in case the condition is false, and the other that consumes the event and invokes the action in case the condition is true. Creating two rules is necessary in order to correctly realize the semantics of ECA-DL, which allows actions to be invoked only when the event occurs, and simultaneously the condition is true. Suppose we would like to specify the behaviour "when John enters the room, and his computer is on". The action should only be invoked when at the time John enters the room (Upon clause), his computer is on (When clause). When John enters the room, and the computer is off (false condition), the action should not be

triggered. Analogously, when John enters the room, the computer is off, and minutes later he turns his computer on, the action should still not be invoked, since at the time the event occurred, the condition was false.

Event consumption is realized with the retract Jess operation, which deletes a particular fact from the working memory. For example, *Table 7-5* depicts the Jess rules generated from a simple ECA-DL rule. *Jess rule 1* checks whether the event occurred, and the condition is false. If this is the case, the event is consumed by retracting if from the Jess working memory. *Jess rule 2* checks whether the event has occurred and the condition is true. If this is the case, the event is consumed and the action is invoked. For ECA-DL rules with no when clause, only one Jess rule is generated, namely Jess rule 2, excluding the (condition) expression. In Appendix C we present some examples of Jess rules which are generated from ECA-DL rules.

*Table 7-5* Jess rules generated from a single ECA-DL rule

| ECA-DL rule | Jess rule 1 | Jess rule 2 |
|---|---|---|
| Upon event<br>When condition<br>Do action | (event)<br>(not condition)<br>=><br>retract (event) | (event)<br>(condition)<br>=><br>retract (event)<br>(action) |

### 7.3.2 Upon clause

The realization of the Upon clause in Jess refers to the realization of event compositions. In the following sections we elaborate on the activities necessary to realize the semantics of event compositions, as we have defined in 7.2.3.

***Event consumption***

In order to implement event consumption, events in the working memory are always related to particular rules. For example, if two rules r1 and r2 refer to the same event e1, we would represent e1 in the working memory twice as r1e1, and r2e1, each serving a particular rule, namely r1 and r2, respectively. This way, rule r1 can consume r1e1, without interfering with rule 2. This mechanism allows rules to consume events without interfering with each other.

Therefore, for each event notification received by the controller component, *n* event facts are asserted in the working memory, *n* being the number of rules that refer to this event. The ECA handler and the event handler components keep track of the rules and events referred by these rules. When an event notification is received, the event handler includes the respective event facts in the Jess's working memory.

*Figure 7-21* depicts an example of sequence of messages exchanged among application components, context processor components and the subcomponents of the controller architecture. An application component, namely application1, subscribes an ECA rule with the controller component, which is received by the ECA handler component. This component invokes the handleEvents operation of the event handler component, passing as argument the collection of events that need to be handled for this particular ECA rule. The event handler component checks for each event description whether there is already a subscription for that particular event. If no subscription is found, an event subscription operation is invoked with the appropriated context processor component (operation (subscribe (transition, characterization)). The appropriate context processor component is found based on the discovery mechanism already explained. Event notification messages are delivered to the event handler whenever the event of interest occurs (operation notify (event)). Upon receiving an event notification, the event handler checks which rules refer to that particular event being notified. The event handler invokes an assert operation for each pair (rule, event), in each rule that refers to the event.

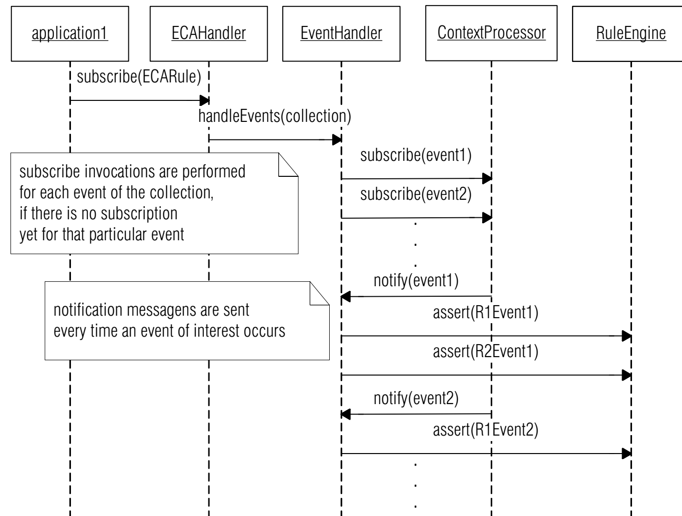



*Figure 7-21* Sequence diagram that illustrates event subscriptions

Events have unique identifiers, which are assigned by the event handler component. Parameterized situation and primitive events are considered as different events, and therefore, are given different unique identifiers. For example, situation event EnterTrue (SituationContained (Person.John)) is different from EnterTrue (SituationContained (Person.John, Building.Zilverling)), which is different from EnterTrue (SituationContained (Person.Mary)). Therefore, these events are referred to by using different unique identifiers.

***Event facts***

Primitive, situation and temporal events are represented in the Jess working memory using event structures that reflect the definition of these events as in the measurement datatypes (Chapter 5). In addition, in order to allow proper event consumption, we include in this event structure the unique identifier of the event, the unique identifier of the ECA rule referring to that particular event, and a timestamp that refers to the time an event is detected. *Figure 7-22* depicts the event types representing some examples of event structures. Suppose three types of events have been defined as part of the context and situation information modelling phase (Chapter 5), namely (EnterTrue and EnterFalse) SituationContainedEvent, (EnterTrue and EnterFalse) SituationWithinRangeEvent, and IncomeCall. The parameters of these events are defined as attributes of the classes depicted in *Figure 7-22*. The SituationContainedEvent event, for example, in addition to the common attributes inherited from class Event, defines the attributes transition, containerID, containedID. The transition attribute defines whether this is an EnterTrue or an EnterFalse event. The containerID and containedID attributes define the parameters of the event notification, which are the entities involved in this particular situation transition. These event structures are reflected in the Jess working memory using shadow facts.

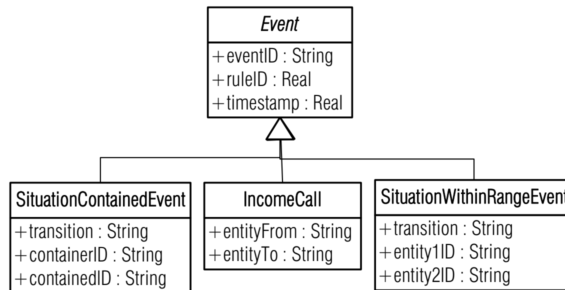



*Figure 7-22* Event types used in the realization phase

***Event composition***

As discussed in the previous chapter, event facts are represented in the Jess working memory using the shadow facts mechanism, which creates shadows of the Java event objects into the working memory. If we look in the Jess working memory, we would see examples of event facts, such as follows:

```
(SituationContainedEvent (eventID “ev1”) (ruleID “rule1”) (timestamp 12312)
        (transition “EnterTrue”) (containerID “JohnHouse”) (containedID “John”)
(IncomeCall (eventID “ev2”) (ruleID “rule1”) (timestamp 142125)
        (entityFrom “John”) (entityTo “Mary”)
```

The identifier of an event and the identifier of an event fact are slightly different, because of the event consumption mechanism we use. We define

that an event fact is uniquely identified using values, namely eventID and ruleID.

*Table 7-6* depicts how the event composition operators can be mapped to Jess expressions. We use the timestamp slot value to verify the temporal occurrence of events. For example, the operation e1;e2 verifies whether e1 occurred before e2 by checking their respective timestamp values. The timestamp value captures the time the event occurs in milliseconds, more precisely, it represents the number of milliseconds since January 1, 1970, 00:00:00 GMT until the date of the event occurrence.

*Table 7-6* Mapping event composition to Jess

| Event expression | Jess expression |
|---|---|
| e1&e2 | (EventType1 (eventID "e1") (ruleID "r1"))<br>(EventType2 (eventID "e2") (ruleID "r1")) |
| {e1;e2} ! e3 | (EventType1 (eventID "e1") (ruleID "r1") (timestamp ?t1))<br>(EventType2 (eventID "e2") (ruleID "r1") (timestamp ?t2&:(> ?t2 ?t1)))<br>(not (EventType3 (eventID "e3") (ruleID "r1") (timestamp ?t3&:(and (>?t3 ?t1) (<?t3 ?t2))))) |
| e1 \| e2 | (or (EventType1 (eventID "e1") (ruleID "r1"))<br>(EventType2 (eventID "e2") (ruleID "r1"))) |
| e1;e2 | (EventType1 (eventID "e1") (ruleID "r1") (timestamp ?t1))<br>(EventType2 (eventID "e2") (ruleID "r1") (timestamp ?t2&:(> ?t2 ?t1))) |

As we have discussed, events are only considered for detecting composite events if they occur within the *detection window interval*. In order to implement the detection window interval in Jess, we include an extra Jess rule for each ECA rule in the controller. This extra rule detects and discards event facts that fall out the detection window interval. In order to do that, we check whether the timestamp of an event plus the detection window interval value (also in milliseconds) is greater than the current time. In order to get the current time value in milliseconds we use the Java System's operation currentTimeMillis. In order to guarantee that the detection window interval rule is executed before the other rules, we give priority to this rule (salience 2). Considering that the detection window interval value is *dw*, and that ECA rule "r1" refers to all the events depicted in *Figure 7-22*, the detection window interval rule of "r1" is:

```
(defrule DetectionWindowECAr1
(declare (salience 2))
?eventfact <-
    (or (SituationContainedEvent (ruleID "r1")
                              (timestamp ?t&:(> (+ ?t dw) (call System currentTimeMillis))))
        (IncomeCall (ruleID "r1")
```

```
                              (timestamp ?t&:(> (+ ?t dw) (call System currentTimeMillis))))
        (SituationWithinRangeEvent (ruleID "r1")
                              (timestamp ?t&:(> (+ ?t dw) (call System currentTimeMillis)))))
=>
(retract ?eventfact))
```

For example, suppose we would like to map the following Upon clause (of ECA rule rule1) to Jess expressions:

```
Upon EnterFalse (SituationContained (Person.John, Building.HouseJohn)) ;
     EnterFalse (SituationContained (Person.Mary, Building.HouseJohn))
```

This Upon clause is evaluated to true when John leaves his house, followed by Mary. The interval between John leaving and Mary leaving should be within the detection window interval. These events are given unique identifiers by the event handler component. Suppose the following unique identifiers are assigned as follows:

```
event1: EnterFalse (SituationContained (Building.HouseJohn, Person.John))
event2: EnterFalse (SituationContained (Building.HouseJohn, Person.Mary))
```

Therefore, the corresponding Jess expression according to *Table 7-6* would be the following:

```
(SituationContainedEvent (eventID "event1") (ruleID "rule1") (containerID "HouseJohn")
                         (containedID "John") (timestamp ?t1))
(SituationContainedEvent (eventID "event2") (ruleID "rule1") (containerID "HouseJohn")
                         (containedID "Mary") (timestamp ?t2&:(> ?t2 ?t1)))
```

### 7.3.3 When clause

Mapping a *when expression* to a Jess expression is similar to mapping an OCL invariant expression to a Jess expression, which has been extensively discussed in Chapter 6. According to the ECA-DL metamodel depicted in *Figure 7-18*, a when clause can be a *binary composite condition*, a *unary composite condition*, a *term* or a *select*. Details for each of these elements are discussed separately in Appendix C.

***Term, binary and unary expressions***

Navigating through context and situation models in ECA-DL is similar to the navigation used in the OCL language, except that we have created some *shortcuts* to facilitate the specification of commonly used constraints. For example, when we would like to refer to a given entity in ECA-DL, e.g., entity1, we use the expression Entity1Type.entity1, as we have elaborated in

section 7.2.2. This corresponds to the OCL expression Entity1Type.allInstances()->any (id = "entity1"), which selects all instances of type Entity1Type, and returns the one whose attribute id is "entity1". In Jess this would be defined as (EntityType1 (OBJECT ?object) (id "entity1")), which is a pattern that matches the entities of type EntityType1 whose slot id has value "entity1". *Table 6-2* (Chapter 6) shows examples of mappings from OCL to Jess, which can also be used as reference for the mapping of navigation between ECA-DL and Jess. Appendix C provides examples of mappings from ECA-DL terms to the Jess language.

Logical binary composite conditions (using the and, and or operators), and unary composite conditions (using the not operator) are mapped to the Jess pattern matching operators as shown in Appendix A. For example, the ECA-DL expression operand1 and operand2 maps to expression (operand1) (operand2) in Jess. Similarly operand1 or operand2 in ECA-DL maps to expression (or (operand1) (operand2)) in Jess. Furthermore, occurrences of and expressions within or expressions, such as ((operand1 and operand2) or operand3), are mapped to (or (and (operand1 operand2)) (operand3)) in Jess. Other binary composite conditions using equality or other comparison operators (e.g., =, >, <) are mapped as shown in Appendix A. Appendix C depicts some examples of how binary composite conditions in ECA-DL can be mapped to binary expressions in Jess.

### *Select expression*

An ECA-DL select expression is mapped to a Jess *defquery* expression. A defquery returns a collection of facts in the working memory, which match a number of conditions. A defquery is declared separately from a rule, and has the following format:

```
(defquery <name-defquery>
    (declare (<variables>))
    (<expressions>))
```

The *run-query* command is used to invoke a defquery and to supply values for the external variables of a query and obtain a list of matches. Consider the following defquery, which returns a collection of fact Persons whose age is above a certain value, which is provided as an argument for the query.

```
(defquery getPersonsAboveAge
   (declare (variables ?value))
   (Person (age ?age&: (>?age ?value)))
)
```

In order to run this query, the Jess command (run-query getPersonsAboveAge 30) should be used. In addition, the Jess command count-query-results can be used to obtain the number of matches of a query. Consider for example the following ECA-DL select clause.

```
Select (Policeman.*, p2; SituationWithinRange (p1, p2) or SituationWithinRange (p2, p1))
```

This select clause returns a collection of policemen that are within range of policeman p1, which is a variable that has been previously defined (for example, in a scope clause). In order to generate a defquery from this select clause, the first argument of the select clause, which is the collection Policemen.*, is mapped to a pattern match of all facts of that particular entity type. In addition, we assign the value of the OBJECT slot to the select variable, which is p2 in the given example. This would be the expression (Policemen (OBJECT ?p2)) in Jess. Mapping the condition expression (the last argument of the select clause) to Jess is similar to mapping any ECA-DL condition expression to a Jess expression. The following defquery corresponds to the aforementioned select clause:

```
(defquery getPolicemenWithinRange
   (declare (variables ?p1))
   (Policemen (OBJECT ?p2))
     (or (SituationWithinRange (person1 ?p1) (person2 ?p2))
        (SituationWithinRange (person1 ?p2) (person2 ?p1))
     ))
```

The ECA-DL function count (collection) can be mapped to the count-query-results Jess command. For example, count(Select (Policeman.*, p2; SituationWithinRange (p1, p2) or SituationWithinRange (p2, p1))), can be mapped to (count-query-results getPolicemenWithinRange ?p1), which returns the number of policemen that are within range of policemen ?p1.

### 7.3.4 Do clause

The do clause of an ECA-DL rule consists of a sequence of action function invocations. As already discussed in Chapter 7, function invocations are mapped to the Jess call command. The arguments to be passed to an action should be previously resolved in the LHS of the rule. Consider the following action clause:

```
Do NotifyApp (application-address,
        List (p1, Select (Policeman.*, p2; SituationWithinRange (p1, p2) or
                          SituationWithinRange (p2, p1))))
```

Both functions NotifyApp (action function), and List (auxiliary) function are implemented in the Java application, rather than in Jess. The defquery getPolicemenWithinRange implements this select clause in Jess. The following RHS is generated in Jess:

```
(… LHS…)
=>
(call SomeClass NotifyApp “application-address”
        (call SomeClass List “id1” (run-query getPolicemenWithinRange ?p1)))
```

### 7.3.5 Scope clause

In general, the scope clause maps to a simple pattern match, in which the value of the OBJECT slot is assigned to the scope variable. The select clause nested in as scope clause is not evaluated with a defquery, as explained in the previous section. Instead, the select expression is mapped to condition on slot values, following the mapping of a normal ECA-DL condition expression. For example, consider the following example of scope clause, which selects all persons located in “Enschede”, and assigns variable p to each of these persons:

```
Scope (Select (Person.*; person; person.hasCivilLocation.city = “Enschede”); p))
```

The corresponding Jess expression pattern matches the persons located in Enschede, and assigns the value of slot OBJECT to variable ?p. This variable can then be used in other parts of this rule.

```
(Person (OBJECT ?p) (hasCivilLocation ?hasCivilLocation))
(CivilLocation (?hasCivilLocation) (city “Enschede”))
```

The following example of scope clause selects all persons that are within range of a person whose id is id1, and assigns variable p to each of these persons:

```
Scope (Select (Person.*; person; SituationWithinRange (person, Person.id1); p))
```

The corresponding Jess expression is as follows:

```
(Person (OBJECT ?p))
(Person (OBJECT ?pid1) (id id1))
(SituationWithingRange (person1 ?p) (person2 ? pid1))
```

The scope clause introduces a problem for the Upon clause when primitive and situation events refer to a scope variable, which is related to how event subscriptions are maintained. Suppose our scope clause defines a variable p, like the example presented above, i.e. p represents each person that is within range of person whose id is id1. Consider now that a nested upon clause defines an event, such as EnterTrue (SituationContained ( ,p)), which generates event notifications every time person p enters any container entity. As we have mentioned, an Upon clause generates event subscriptions with context sources and managers. Event subscription for an Upon clause without scope is trivial, and it has been discussed in section 7.3.2. However, event subscriptions for an Upon clause nested in a scope clause requires special attention, since the collection defined in the Upon clause is dynamic, i.e. the entities belonging to the scope collection change over time. This means that the persons within range of person id1 may change over time, and to reflect that, new event subscription may need to be performed, and old event subscriptions may need to be cancelled while the rule is still executing.

In order to overcome this problem, we create a separate Jess rule that detects every time there is a change in the collection resulted from a scope clause. When this collection changes, we invoke a method of the event handler component, namely maintainSubs to indicate that there was a change in the scope collection. Based on these indications, the context handler may perform new event subscriptions or may cancel old ones that are no longer needed. The argument passed to the maintainSubs method is the result of a defquery, which should be defined from the scope nested in the select clause. For the SituationWithinRange scope example, the following defquery should be defined:

```
(defquery PeopleWithinRange
   (declare (variables ?p1))
   (Person (OBJECT ?p2))
   (SituationWithinRange (entity1 ?p1) (entity2 ?p2)) )
```

In addition, the following Jess rule should be defined, in order to invoke the maintainSubs method:

```
(defrule SubscriptionScopeRule
  (Person (OBJECT ?p))
  (Person (OBJECT ?pid1) (id id1))
  (SituationWithingRange (entity1 ?p) (entity2 ?pid1))
  =>
  (call EventHandler maintainSubs "ecarule1" (run-query PeopleWithinRange ?pid1))
```

The event handler component performs and maintains event subscription registrations for each of the entities in the scope. This way, event notifications that are received for all entities in the scope are correctly included into the working memory. Suppose, for example, that persons John, Mary and Alice are currently supported, and that John is within range of Mary and Alice. Consider the following ECA-DL rule (rule3):

```
Scope (Select (Person.*; person; SituationWithinRange (person, Person.John); p))
{
    Upon IncomeCall (p, Person.John)
...}
```

According to this rule, the event handler component should maintain registrations of IncomeCall event notification from both Mary and Alice to John, since they are both in the scope. *Table* 7-7 depicts the registration structure that should be maintained by the event handler component.

*Table 7-7* Example of event registrations that are maintained in the event handler component

| Event Description | ruleID | Parameters |
|---|---|---|
| IncomeCall | rule3 | Mary, John |
| IncomeCall | rule3 | Alice, John |

When an IncomeCall event notification is received, for example IncomeCall (Peter, John), the controller matches this notification with the registrations available (*Table* 7-7). Since no matches are found, the event handler does not include the event notification in the Jess working memory. Suppose now that the event notification IncomeCall (Mary, John) arrives. Since now there is a match, an instance of IncomeCall event notification is created in the Jess working memory.

So far, we have solved the issue of subscribing to event notifications dynamically. We have not yet discussed how events are consumed by an scoped ECA-DL rule. Similar to the rules with no scope clause, we assign a variable to the matched event fact on the rule's LHS, in such way we can retract this event fact on the RHS of the rule, as follows:

```
?eventfact <- (IncomeCall (eventID "ev1") (ruleID "rule3") (entityFrom ?p)
              (entityTo "John"))
(... other conditions ...)
=>
(retract ?eventfact)
(... actions...)
```

### 7.3.6 Loading the working memory

For the correct execution of the rules we have presented so far, the working memory should be always up-to-date. The synchronization between Java objects and the working memory in the controller component is based on shadow facts, as discussed in Chapter 6. Therefore, we should guarantee that the Java objects contain up-to-date context and situation values every time the Jess engine needs to check the conditions defined in the when clause of a rule. Keeping the working memory up-to-date with respect to events in the *Upon* clause is carried out by subscribing to events, as discussed in section 7.3.2. With respect to the *When* clause, we consider two different approaches, namely *subscription-based* or *query-based*.

The subscription-based approach is realized at the time an ECA rule is subscribed with the ECA handler component. The ECA handler component checks the When clause, and identifies which context and situation values need to be gathered from context sources and managers, respectively. The ECA handler component then indicates the event handler component the list of required context and situation information values.

For context value types, time-based subscriptions may be performed. For example, if the when clause refers to Person.John.hasGeoLocation.coordinates.latitude, the event handler component needs to find a context source that is capable of offering instances of John's location context (GeoLocation information). When that particular context source is found, the event handler component subscribes to it in order to receive time-based notifications with updates of John's location. These notification values generate updates to the Java objects, which guarantee that the working memory is up-to-date.

For situation value types, event-based subscriptions are performed, since the controller components wants to receive notifications about changes in the situation. For example, if the When clause refers to SituationContained (Building.JohnHouse ,Person.John), the event handler component subscribes to a context manager component which is capable of offering this situation value. In this case, EnterTrue and EnterFalse event notifications are expected. The EnterTrue event notification creates a situation instance in the working memory, and EnterFalse event notification deactivates that particular situation instance in the working memory.

*Figure 7-23* depicts a sequence diagram that illustrates time-based subscriptions for the GeoLocation example. The event handler subscribes with a context source to receive time-based notification with the value of John's geographical location. When these notifications are received, the respective GeoLocation Java object is updated accordingly.

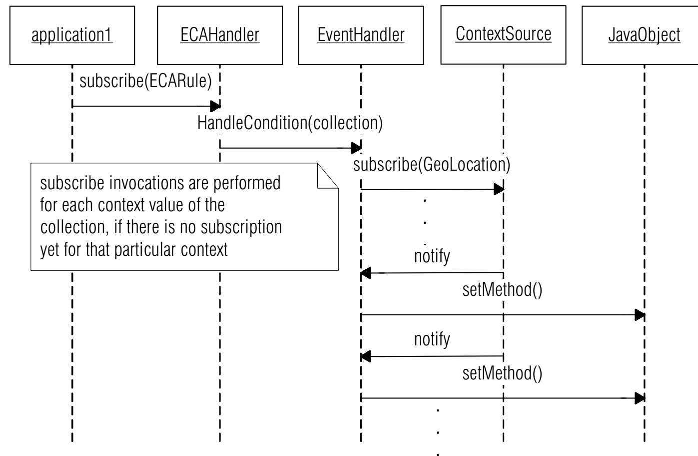


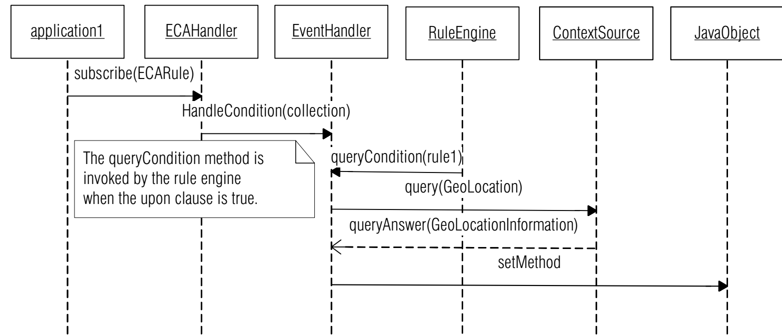

*Figure 7-23* Sequence diagram for time-based subscriptions

The query-based approach is realized only when the events in the Upon clause have occurred. Contrary to time-based and event-based approaches, the query-based approach is not realized upon an ECA rule subscription. Instead, context source and managers components are queried when the rules are already running in the rule engine. In order to realize this approach, we include an extra Jess rule that indicates to the event handler component when the events in the Upon clause occur. The execution of this rule has higher priority than the Jess rules normally generated from an ECA rule (see *Table 7-5*), but it has lower priority than the detection window interval rule. Therefore, we declare salience 1 to this rule, which gives priority over normal rules, but not to the detection window interval rule, which has salience 2. The general format of this rule is the following:

```
(defrule queryConditionRule1
  (declare (salience 1))
  (event)
  =>
  (call EventHandler queryCondition rule1))
```

The event handler component keeps the list of context and situation values that need to be queried. The queryCondition method implemented by the event handler component starts the query process. The result of the queries generates updates in the working memory by means of shadow facts. *Figure 7-24* depicts an example of query-based approach for the geographical location example. The event handler component only queries the context source when the queryCondition method has been invoked. Query answers generate set method invocations on the Java objects, which updates the working memory.

Figure 7-24 Sequence diagram for query-based approach




### 7.3.7 Automating the mapping from ECA-DL to Jess

We have automated the generation of Jess rules from ECA-DL rules using two different alternatives, namely by building a *parser* and by defining *MDA transformations*. Automation allows Jess code to be generated automatically from ECA-DL rules, with no need to write code by hand. Both automation processes should consider the systematic derivation of Jess code from ECA-DL rules as we have discussed in the previous sections.

#### *Parser approach*

*The parser* approach implements a parser for ECA-DL rules that generates Jess rules using a simplified version of the mappings discussed so far [33, 37]. This parser receives ECA-DL rule descriptions in XML format, rather than text. In order to allow XML documents to be checked for conformance with respect to the ECA-DL syntax, we have created an XML schema, which reflects the ECA-DL syntax presented in section 7.2.4. A simplified version of this schema can be found in [28].

As soon as an application subscription is received by the ECA handler component, it is parsed and each element of this rule is identified. In order to be a *valid* rule, each of these elements in the rule should exist in the context and situation model adopted by the controller component. The parser generates a tree composed by the primitive elements of ECA-DL. For example, the ECA-DL rule below has its parsed tree depicted in *Figure 7-25*. Variable definitions and their respective reference variables are depicted using the same line pattern.

```
Scope (Select (Policeman.*; policeman; policemen.hasActivity.value = ‘working’; p1)
{ Upon OnEvery (5)
  Do NotifyApp (application-address,
     List (p1.id, select (policeman.*, p2; SituationWithinRange (p1, p2) and
             p2.hasActivity.value = ‘working’)))}
```

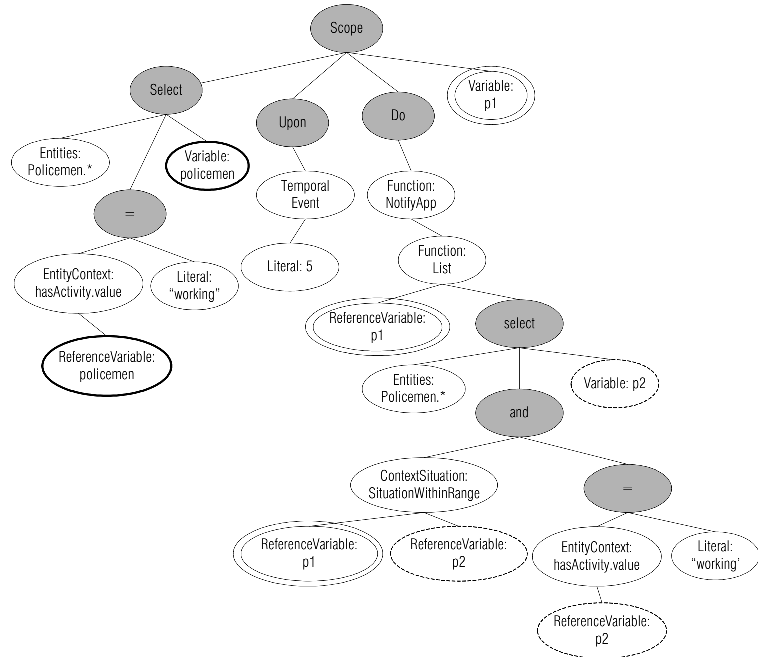



*Figure 7-25* Example of parsed ECA-DL rule

The parser performs two levels of validity checking of an ECA-DL rule:

- A typing and static semantics checking, using the context and situation models adopted by the controller component. At this level, the parser verifies the existence of the entity, context and situation types, and the proper combinations between (i) entity types and context types; and (ii) between situation types and entity types. For example, the combination SituationDriving (Building.id) should give an error because the situation driving is not applicable to an entity of type Building. In addition, the parser verifies the validity of Functions and Actions. Specific functions and actions have some number and types of parameters defined when they are specified. For example, function sendSMS receives as argument the identifier of a person, and a message to be sent to the person's mobile phone.
- An instance checking level to check the existence of specific entities in the controller component. If an application ECA-DL rule refers to entities Person.id1, Person.id2, Building.id3, and Building.id4, the parser checks the existence of such entities in the current Java implementation of the context model.

Once an ECA-DL rule is validated, the parser generates Jess expressions from each of the parsed ECA-DL elements, following the mappings we have described in the previous sections.

### *MDA approach*

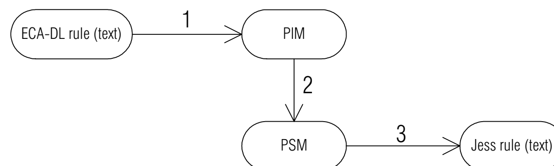
We have also worked on an alternative solution [75] to generate Jess rules from ECA-DL, which is based on the Model Driven Architecture (MDA) [92] approach. The aim of this solution is to *formalize* the mapping process from ECA-DL rules to Jess rules and implement this mapping following the MDA approach.

The main idea behind MDA is that applications are no longer architected to specific platforms such as Java, or C#, but instead are specified as *Platform-Independent Models (PIM)*. These models allow the design of application-specific concerns without regard for platform-specific concerns. Such platform-independent model can be transformed to a *Platform-Specific Model (PSM)*, which considers platform-specific concerns. This PSM can then be transformed to an actual implementation, possibly using automated transformations.

This design approach facilitates (i) the maintenance of rules when a newer version of the Jess engine is released; and (ii) the implementation of ECA-DL rules in a different rule-based platform, e.g., CLIPS or Mandarax. Adopting a different or an improved platform requires regenerating the PSM and actual implementation, avoiding costly manual conversions, in case automated transformations are used.

Following the MDA approach, ECA-DL and Jess rules are no longer treated as text, but are converted to models, and the transformation between rules occurs at the model level. We have followed the Meta-Object Facility (MOF) [93] standard defined by the OMG, which specifies the modeling architecture we applied. In this architecture, the first step towards the model-based transformation process is to define the language metamodels, and how each element of the ECA-DL metamodel relates to the Jess metamodel. The transformation from elements of the ECA-DL metamodel to elements of the Jess metamodel is specified in a *transformation language*. We have used the ATLAS Transformation Language (ATL) [3, 4] for specifying transformations from ECA-DL to Jess, since there is available tool support for this language for the Eclipse development environment [36]. *Figure 7-26* depicts an overview of the transformation process.

*Figure 7-26* Overview of the transformation process




In step 1 (*Figure 7-26*), we define an instance model of an ECA-DL rule, which contains entity, context and situation instances, conforming to the ECA-DL metamodel. This step is performed using text-to-model

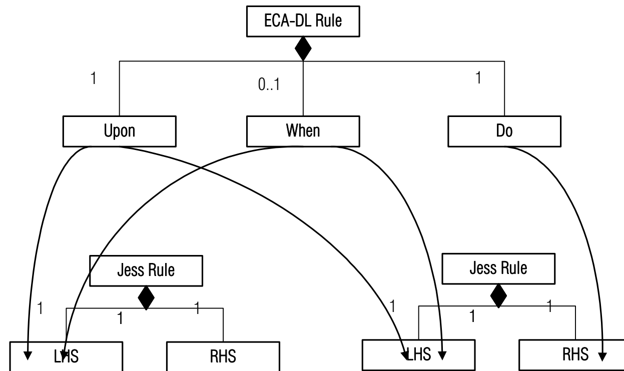
transformation. In MDA terminology, the resulting model is a PIM model, since ECA-DL is the platform-independent language in this particular case.

Step 2 concerns the transformation from the PIM to a PSM, which is done (semi)-automatically by a transformation tool. For the automated part of this model-to-model transformation, the OMG has specified the *Queries/Views/Transformations (QVT) standard* [107]. *Queries* can be used to access the source model and extract information from it, *Views* can be used to focus on specific aspects, and *Transformations* can be used to take the information from the Queries and produce a target model. The result of this transformation is a PSM model, since Jess is the platform-specific language in this case.

In step 3, the PSM is transformed back to its textual form, ready for execution in the Jess engine. This step is performed using model-to-text transformation.

*Figure 7-27* depicts a fragment of the mappings between the ECA-DL and the Jess metamodels. The arrows here indicate which part of the ECA-DL rule is mapped to which part of the Jess rules. As shown in *Table 7-5*, an ECA-DL rule (without scope clause) generates two Jess rules. Both the Upon and When clauses are mapped to two LHS's in Jess. The Do clause maps to the RHS of the second Jess rule. The Jess parts that are not linked to the ECA-DL rule are filled in by the transformation process. For details of the transformation specification in ATL, we refer to [75].

*Figure 7-27* Fragment of the mapping between the ECA-DL and Jess metamodels




## 7.4 ECA rules and the Situation Detection Framework

We have discussed through this chapter how the controller component communicates with context manager components to obtain situation information (sections 7.3.2 and 7.3.6). Our discussions have been based on service-oriented architectures, and the messages exchanged between the

controller and the context manager components follow the event-based interaction style for both upon and when clauses.

An alternative solution for communicating with context manager components for obtaining situation information is using some rule-based middleware to allow distribution, such as the DJess middleware, discussed in section 6.5.3. DJess allows Jess engines running on different nodes of a network to communicate. It provides transparencies for context and situation distribution, i.e. each engine works on the distributed facts as if they were local. Using the DJess middleware, the controller component can become a member of the DJess *web of inference*, and the situation facts detected on remote nodes can be automatically reflected in the controller's working memory.

*Figure 7-28* depicts a distribution scenario, in which two situation detection engines and a controller's rule engine participate in a web of inference. ECA-DL rules in the controller's rule engine execute on the contents of the controller's working memory, which is (partially) shared with the other engines.

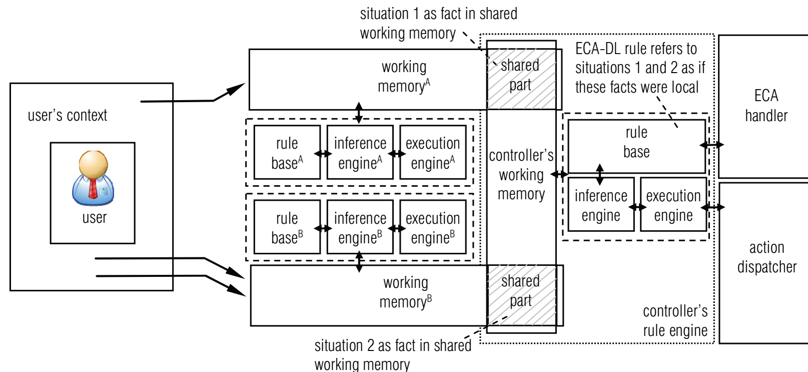



*Figure 7-28* Using DJess to realize the controller's rule engine

Suppose an ECA-DL rule refers to situation facts 1 and 2, which are detected by remote engines, namely, engines A and B, respectively. Since the controller and the engines A and B participate in the same web of inference, situation facts 1 and 2 are also perceived by the controller's working memory. The benefit of using DJess is that it alleviates the responsibility of managing event-subscription messages and query-based messages. Therefore, the design and implementation of the controller component is substantially simplified.

Since situation event notifications are no longer expected by the controller component, the way the Upon and Scope clauses are realized in DJess differs from the way they are realized in Jess. In the service oriented approach, event notifications are identified by a unique identified, which is assigned by the event handler component. When the event handler receives

a notification, it includes the respective event into the controller's working memory. Suppose, for example, we would like to detect the following upon clause of rule1: Upon EnterTrue (SituationContained (Zilverling, John)); EnterTrue (SituationContained (HouseJohn, John)), which specifies that John enters the Zilverling building, and after that he enters his house. The event handler assigns the following unique identifiers to these events:

ev1: EnterTrue (SituationContained (Zilverling, John))
ev2: EnterTrue (SituationContained (HouseJohn, John))

When the event handler component receives notifications of these events, it creates corresponding instances of SituationContainedEvents, which are shadowed in the working memory. Using DJess this mechanism works differently, since there is no event handler component intermediating the working memories. Situation facts detected in a particular working memory are perceived by the others participating in the same web of inference. In order for the controller component to detect the occurrence of situation events (event facts), we include additional DJess rules in the controller's rule engine that explicitly detect and assert these event facts. *Table 7-8* depicts the rules that are used to detect events for a particular ECA-DL rule in DJess.

*Table 7-8* Examples of Jess rules to detect situation events

| Type of rule | DJess rule |
|---|---|
| ECA-DL Upon clause | Upon EnterTrue (SituationContained (Zilverling, John));<br>EnterTrue (SituationContained (HouseJohn, John)) |
| DJess rule for event ev1 | ;enterTrue<br>(defrule event1Rule1<br>(SituationContained (person "John") (building "Zilverling") (finaltime nil))<br>=><br>(assert (SituationContainedEvent (eventID ev1) (ruleID rule1) (transition "EnterTrue") (containerID "Zilverling") (containedID "John") ...)) |
| DJess rule for event ev2 | ;enterTrue<br>(defrule event2Rule1<br>(SituationContained (person "John") (building "HouseJohn") (finaltime nil))<br>=><br>(assert (SituationContainedEvent (eventID ev2) (ruleID rule1) (transition "EnterTrue") (containerID "HouseJohn") (containedID "John") ...)) |

If we would like, for example, to detect when John leaves the Zilverling building, i.e. Upon EnterFalse (SituationContained (Zilverling, John)), the event detection rule would verify whether the situation final time is not nil, which

indicates that the event EnterFalse has occurred for this particular situation. The DJess rule would be the following:

```
; enterFalse
(defrule event3Rule1
  (SituationContained (person "John") (building "Zilverling") (finaltime ~nil))
=>
  (assert (SituationContainedEvent (eventID ev3) (ruleID rule1) (transition "EnterFalse")
                                   (containerID "HouseJohn") (containedID "John") ...))
```

These additional rules detect when particular situation events occur, and insert corresponding event facts in the working memory. These rules are included when an ECA-DL rule is subscribed with the controller component. For each situation event referred in an ECA-DL rule, a DJess rule should be defined. These rules assert the corresponding event fact into the working memory.

In order to realize the scope clause in DJess, we also need to modify the approach discussed in section 7.3.2, since there is no need to maintain situation event subscriptions for the scope clause, except for primitive events, which are still expected by means of primitive event notifications. Therefore, we do not need to create the Jess rule that maintains the scope subscriptions up-to-date, namely, we do need to include the SubscriptionScopeRule rule in order to implement the scope clause. Suppose for example, the following scope clause is defined in an ECA-DL rule:

```
Scope (Select (Person.*; person; SituationWithinRange (person, Person.id1); p))
{
  Upon EnterFalse (SituationContained (Building.HouseJohn, p))
  When conditions
  Do actions
}
```

Three DJess rules should be defined to implement this ECA-DL rule (see Table 7-9):

- A scope DJess rule that detects the occurrences of event EnterFalse (SituationContained (Building.HouseJohn, p)), where p is any person within range of person whose id is id1;
- A *DJess rule1* that consumes the event in case the event occurs and condition does not hold;
- A *DJess rule2* that consumes the event and triggers the action when both the event occurs, and the condition holds.

*Table 7-9* DJess rules that implement an ECA-DL rule with scope clause

| Type of rule | DJess rule |
|---|---|
| Scope DJess rule | ;enterFalse<br>(defrule event1Rule1<br>(SituationWithinRange (person ?p) (person2 "id1"))<br>(SituationContained (person ?p) (building "Zilverling")<br>(finaltime ~nil))<br>=><br>(assert (SituationContainedEvent (eventID ev1) (ruleID rule1) (transition "EnterFalse") (containerID "Zilverling") (containedID ?p)...)) |
| DJess rule1 | (defrule DJessRule1<br>?evenfact <- (SituationContainedEvent (eventID ev1) (ruleID rule1)<br>(transition "EnterFalse") (containedID ?p))<br>(not conditions)<br>=><br>(retract ? evenfact)) |
| Djess rule2 | (defrule DJessRule2<br>?evenfact <- (SituationContainedEvent (eventID ev1) (ruleID rule1)<br>(transition "EnterFalse") (containedID ?p))<br>(conditions)<br>=><br>(retract ? evenfact)<br>(actions)) |

Consider now the following ECA-DL rule example, in which two events are composed in the Upon clause. This upon clause is true when person p leaves John's house and after that enters the Zilverling building. Scope variable p refers to the persons within range of person whose id is id1.

```
Scope (Select (Person.*; person; SituationWithinRange (person, Person.id1); p))
{
   Upon EnterFalse (SituationContained (Building.HouseJohn, p)); EnterTrue (SituationContained
(Building.Zilverling, p))
...
}
```

For this ECA-DL rule, four DJess rules should be defined:

– A scope DJess rule1 that detects the occurrences of event Upon EnterFalse (SituationContained (p, Building.HouseJohn)), where p is any person within range of person whose id is id1;

- A *scope Djess rule2* that detects the occurrences of event EnterTrue (SituationContained (p, Building.Zilverling)), where p is any person within range of person whose id is id1;
- A *DJess rule1* that consumes the event in case the event occurs and condition does not hold;
- A *DJess rule2* that consumes the event and triggers the action when both the event occurs, and the condition holds.

*Table 7-10* DJess rules to implement an ECA-DL rule with scope clause and event composition

| Type of rule | DJess rule |
|---|---|
| Scope DJess rule1 | ;enterFalse<br>(defrule event1Rule1<br>(SituationWithinRange (person ?p) (person2 "id1"))<br>(SituationContained (person ?p) (building "HouseJohn") (finaltime ~nil))<br>=><br>(assert (SituationContainedEvent (eventID ev1) (ruleID rule1) (transition "EnterFalse") (containerID "HouseJohn") (containedID ?p)…)) |
| Scope DJess rule2 | ;enterTrue<br>(defrule event2Rule1<br>(SituationWithinRange (person ?p) (person2 "id1"))<br>(SituationContained (person ?p) (building "Zilverling") (finaltime nil))<br>=><br>(assert (SituationContainedEvent (eventID ev2) (ruleID rule1) (transition "EnterTrue") (containerID "Zilverling") (containedID ?p)…)) |
| DJess rule1 | ?evenfact1 <-<br>(SituationContainedEvent (eventID ev1) (ruleID rule1) (transition "EnterFalse")<br>(containerID "HouseJohn") (containedID ?p) (timestamp ?t1)<br>?eventfact2 <-<br>(SituationContainedEvent (eventID ev2) (ruleID rule1) (transition "EnterTrue")<br>(containerID "Zilverling") (containedID ?p) (timestamp ?t2)<br>(timestamp ?t2&:(> ?t2 ?t1)))<br>(not conditions)<br>=><br>(retract ?eventfact1)<br>(retract ?eventfact2) |

| Djess rule2 | ?evenfact1 <-<br>(SituationContainedEvent (eventID ev1) (ruleID rule1) (transition “EnterFalse”)<br>(containerID “HouseJohn”) (containedID ?p) (timestamp ?t1)<br>?eventfact2 <-<br>(SituationContainedEvent (eventID ev2) (ruleID rule1) (transition “EnterTrue”)<br>(containerID “Zilverling”) (containedID ?p) (timestamp ?t2)<br>(timestamp ?t2&:(> ?t2 ?t1)))<br>(conditions)<br>=><br>(retract ?eventfact1)<br>(retract ?eventfact2)<br>(actions) |
|---|---|

## 7.5 Discussion

We have presented in this chapter the detailed design of the controller component, which enables ECA rules to be added to the platform at runtime. The controller component implements a rule engine that can efficiently process rules, which are matched against various types of events, context, and situation conditions. When events have occurred and conditions hold, the action part of the rule is executed, which consists of various types of service invocations.

In order to facilitate the specification of context-aware reactive behaviours, we have developed ECA-DL, a domain specific language for context-aware reactivity. With this language application developers specify ECA-DL rules, which are composed of three parts:

- An event part, which allows complex compositions of temporal, primitive and situation events;
- A condition part that allows various combinations of context and situation conditions;
- An action part that that allows the specification of service invocations.

In addition, an ECA-DL rule may be parameterized by means of the *scope clause*, which facilitates the specification of ECA-DL rules to a collection of entities that respect certain (contextual) conditions.

In order to demonstrate the feasibility of ECA-DL rules, we have discussed how the Jess engine can be used as the ECA-DL execution environment. Since only Jess rules are accepted by the Jess engine, we have discussed mappings that can be used to generate Jess rules from ECA-DL rules.

In general, writing context-aware behaviours in ECA-DL requires less programming effort than in the Jess language, since ECA-DL is specific to the context-awareness domain, while the Jess language is a general purpose

language. Often the realization of a single ECA-DL rule in Jess requires several Jess rules. The mapping problems encountered are mainly related to the lack of support to *events* in Jess. Therefore, the mappings we have defined consider issues to correctly implement event consumption and event composition.

We have discussed two approaches for the automation of the mappings from ECA-DL rules to Jess rules, namely the *parser* and the *MDA approach*. The parser approach implements a parser that breaks down an ECA-DL rule into indivisible elements, which are defined in the ECA-DL metamodel. For each of these elements, Jess expressions are generated. The MDA approach generates Jess rules from ECA-DL based on transformations defined in terms of elements of the metamodels of these languages, following the guidelines of the Model Driven Architecture (MDA) [92] approach. The aim of this solution is to *formalize* the mapping process from ECA-DL rules to Jess rules and implement this mapping following the MDA standards. This design approach facilitates (i) the maintenance of rules when a newer version of the Jess engine is released; and (ii) the implementation of ECA-DL rules in a different rule-based platform, e.g., CLIPS or Mandarax. Both automation alternatives partially consider the mappings issues we have discussed in this chapter. Extensions to the automation processes are needed, and are indicated for future work.

Finally, we have discussed in the chapter how the controller component can be integrated to the situation framework discussed in the previous chapter. We discuss how the mapping framework needs to be adapted in order to use DJess. Since in DJess the working memory is shared, there is no need to manage situation event subscriptions and notifications. This facilitates the mappings of issues related to events.

Chapter 8

# Case Study

This chapter demonstrates the *feasibility* of the development approach proposed throughout Chapters 2 to 7 by means of two design examples. The scenarios we have considered for the demonstration are a *healthcare* scenario, and a *policy management* scenario. Since these scenarios deal with different application requirements, we also demonstrate the suitability of our context handling platform to support applications in different domains.

For each of the scenarios, we provide a high level description of the application intended to implement the scenario, and the design process following the development approach proposed. The design process includes the activities (i) context modelling, (ii) context information modelling; (iii) application structural design; (iv) context provisioning services design; and (v) usage of the controlling service. In order to prove feasibility, we build an application prototype for the healthcare scenario that implements the design products obtained from the design activities. With this prototype we are capable of measuring scalability and performance issues on a realistic application. Finally, we provide in this chapter a discussion that analyses our development approach in the light of the requirements presented in Chapter 2.

This chapter is further structured as follows: section 8.1 presents the design of the healthcare application following the design process proposed; section 8.2 presents the design of the policy management application; 8.3 elaborates on the prototype of the healthcare application, and provides measurements obtained from the prototype; finally, section 8.4 discusses our approach in the light of the requirements presented in Chapter 2.

## 8.1 The Healthcare Application

The healthcare epilepsy scenario has been mentioned several times throughout this thesis. For example, in section 1.2 we have presented a

scenario in which *Mr. Janssen*, who is an epileptic patient, is monitored in order to detect or predict epileptic seizures. Upon an epileptic seizure alarm, a number of actions can be taken, such as warning Mr. Janssen of an upcoming seizure, and sending an SMS message to relatives that are currently near him. This scenario is used in the AWARENESS project, and its relevance on improving the quality of life of such patients has been confirmed by *Roessingh research and development*. Roessingh Research and Development [110] is an internationally recognised research institute that contributes to improvements in rehabilitation medicine with particular interests in rehabilitation technology. The healthcare epileptic scenario storyline, already presented in section 1.2, is as follows:

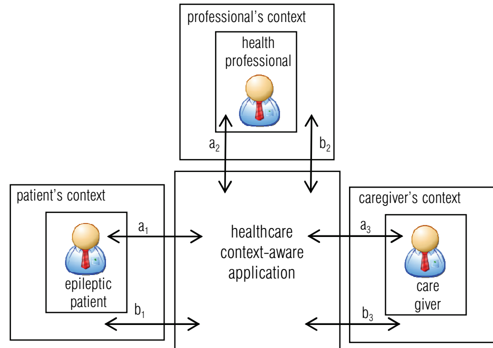
*"Mr. Janssen is an epileptic patient and despite his medications, he still suffers from seizures. Because of his medical condition, Mr. Janssen is unemployed, home-bound, and his situation requires constant vigilance to make sure healthcare professionals are alerted of a severe seizure. Recently, Mr. Janssen has been provided with a tele-monitoring context-aware application capable of monitoring epileptic patients and providing medical assistance moments before and during an epileptic seizure. Measuring heart rate variability and physical activity, this application predicts seizures and contacts nearby relatives or healthcare professionals automatically. In addition, Mr. Janssen can be informed moments in advance about the seizure, being able to stop ongoing activities, such as driving a car or holding a knife. The aim is to provide Mr. Janssen with both higher levels of safety and independence allowing him to function more freely in society despite his disorder."*

In order to implement this scenario, we propose a context-aware application, namely the *healthcare context-aware application*. The aim of this application is to detect the seizures, and to react in the following ways: (i) notify the epileptic patient of an upcoming seizure; and (ii) notify his/her nearby caregivers of an upcoming seizure of the patient by showing a map with the location of the patient. The caregivers who receive the notification for help should be (i) assigned as one of the caregivers of that particular patient; (ii) available for helping; and (iii) physically close to the patient. Upon a notification for help, caregivers may either accept or reject the request for helping the epileptic patient. When a particular caregiver accepts to help, the other caregivers who had received the notification for help are informed that a certain caregiver has already accepted to help that patient.

*Figure 8-1* depicts an intuitive view of the types of users supported in this scenario, their contexts, and the healthcare context-aware application. We focus in this figure on a single instance of user for each type of user supported. *Figure 8-1* is similar to *Figure 2-1* that has been introduced in Chapter 2. Three types of users are shown, namely, the *epileptic patient*, the *healthcare professional* and the *caregiver*. The arrows of type $a_1$, $a_2$ and $a_3$ show that these users and the healthcare application interact. Similarly, the

arrows of type $b_1$, $b_2$ and $b_3$ show that the users' contexts and the healthcare application interact.

*Figure 8-1* Informal view of the healthcare application




For all the three types of users, the interactions of type "a" enable the user's inputs to be provided to the healthcare application, such as starting commands, and defining configuration preferences. Tin addition, the interactions represented by arrows of type $a_1$ enable a patient to receive a notification of upcoming epileptic seizure, which is delivered by the healthcare application. The interactions represented by arrows of type $b_1$ enable the healthcare application to capture particular context conditions from the epileptic patient's context, such as his/her biosignals (e.g., heart rate measures and blood pressure), and location information. An epileptic patient may be performing a potentially hazardous activity, such as holding a knife or driving a car. The interactions represented by arrows of type $b_1$ enable the healthcare application to capture information on whether the patient is performing a hazardous activity or not.

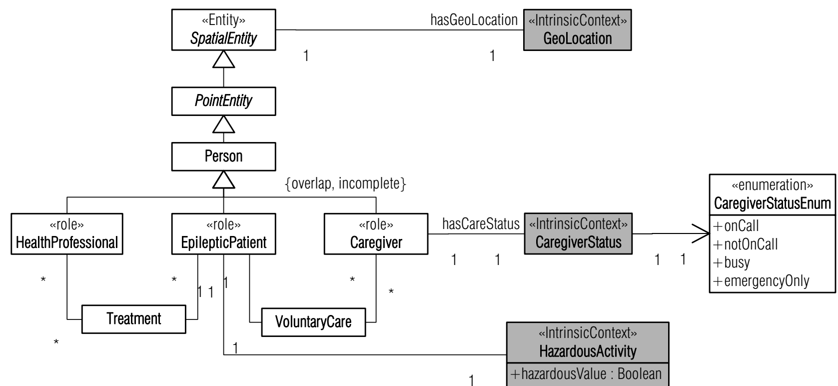
Similarly, the interactions represented by arrows of type $a_2$ enable the health professional's to receive notifications of epileptic seizure alarms and to monitor the biosignals of the patient in order to detect dangerous abnormalities. The interactions represented by arrows of type $b_2$ enable the healthcare application to capture particular context conditions from the health professional's contexts, such as his/her location information.

The interactions represented by arrows of type $a_3$ enable a caregiver to (i) receive notifications for helping particular epileptic patients with upcoming seizures; and (ii) accept or reject requests for help. The interactions represented by arrows of type $b_3$ enable the healthcare application to capture particular context conditions from the caregiver's contexts, such as his/her location and availability status information. Caregiver's availability status might be "on call", "not on call", "busy", or "emergency only", which are explained in detail in the next section.

### 8.1.1 Context modelling

According to our development approach, the first step towards application realization is the modelling of the application's universe of discourse. Our methodology for context and situation modelling has been extensively discussed in Chapter 5. Analysing the application scenario, we identify the entity, context and situation types necessary to model the healthcare application. *Figure 8-2* depicts the context model that represents this application's universe of discourse regarding the relevant entity and context types.

*Figure 8-2* Healthcare application context model




We identify the roles a person can play in this scenario, which are HealthProfessional, EpilepticPatient, and Caregiver. A HealthProfessional can be a doctor, a nurse or a trained person that is capable of providing professional treatment to an epileptic patient. An EpilepticPatient represents the persons who suffer from an epilepsy medical condition. Finally, the Caregiver represents the persons who have volunteered to assist epileptic patients having an epileptic seizure. The relationship between a health professional and an epileptic patient is characterized by a treatment, which is represented by the Treatment class. Similarly, the relationship between an epileptic patient and his/her caregivers is characterized by a voluntary care relation, which is represented by VoluntaryCare class.

Three intrinsic context types are identified: a person's geographical location (GeoLocation), the status of a caregiver (CareStatus), and information on whether a patient is currently doing a hazardous activity (HazardousActivity). Caregivers can set their status to (i) onCall, which specifies they are currently available to receive requests for helping patients, (ii) notOnCall, which specifies they are not available for receiving requests for help, (iii) busy, which specifies they are currently receiving requests, but are busy at the moment; (iv) emergencyOnly, which specifies they are currently available for receiving requests only on emergency situations. A person's geographical

location value is represented by the GeoLocationCoordinates datatype (*Figure 8-4*), which specifies the latitude, longitude and the altitude of the person's current location. In addition, in this datatype we specify the formal relation operations nearness and distance. An epileptic patient may be also doing a potentially hazardous activity, which is captured by a Boolean attribute of the class HazardousActivity.

In order to know whether a person is a caregiver of a given patient, we define a static operation in the Caregiver class (isCaregiver (Caregiver c, EpilepticPatient p)), which receives a caregiver c and patient p as arguments, and returns true if patient p can be helped by caregiver c, and false otherwise. The body of this method is defined in OCL as follows:

```
context Caregiver::isCaregiverOf (caregiver:Caregiver, patient: EpilepticPatient): Boolean
body: caregiver.VoluntaryCare->exists (care| care.EpilepticPatient = patient)
```

Similarly, in order to know whether a person is one of the health professionals who has a treatment relationship with a given patient, we define a static operation in the HealthProfessional class, namely isHealthProfessionalOf (HealthProfessional hp, EpilepticPatient p). This operation receives a health professional hp and patient p as arguments, and returns true if hp is one of the health professional of patient p, and false otherwise. The body of this method is defined in OCL as follows:

```
context HealthProfessional:: isHealthProfessionalOf (professional: HealthProfessional,
                                                      patient: EpilepticPatient): Boolean
body: professional.Treatment->exists (prof| prof.EpilepticPatient = patient)
```

The epileptic seizure alarm is generated by devices attached to the patient's body. These devices collect patient's biosignals in order to predict an epileptic seizure. Roessingh Research and Development has worked on a prediction algorithm that detects seizures based on certain heart rate and blood pressure variation patterns. Epileptic alarms are generated automatically from biosignals, but patients are capable of turning off the alarm in case of an erroneous prediction.

The detection of epileptic seizures generates seizure alarms, which can be defined as primitive events. Epileptic alarm events are represented by (EpilepticAlarm (EpilepticPatient)), and the event generated by the patient when he turns off the alarm is represented by (RejectEpilepticAlarm (EpilepticPatient)). When caregivers who are near the patient receive a notification that a certain patient may be in need of help, a caregiver may accept or reject the request. Acceptance or rejection generates primitive events, which are originated from the caregivers' devices. These primitive events are

represented by AcceptHelpRequest (EpilepticPatient, Caregiver), and RejectHelpRequest (EpilepticPatient, Caregiver), respectively.

Following the design process, application's particular state-of-affairs of interest are modelled, by means of situation models. We define two situation types, which are of interest to the epileptic scenario, namely SituationCaregiverAvailable, and SituationCaregiverWithinRange. The situation type SituationCaregiverAvailable specifies that caregivers are available when their status is set to "onCall" or "emergencyOnly". *Figure 8-3* depicts the specification of this situation using our situation modelling approach.

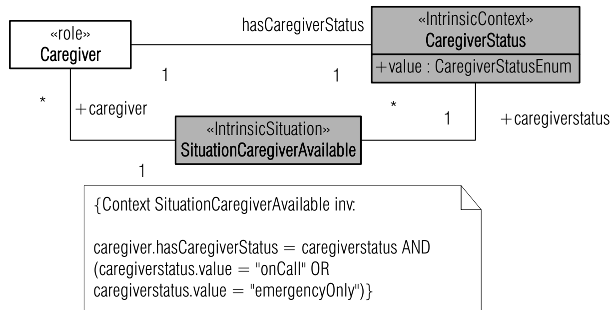



*Figure 8-3* Situation SituationCaregiverAvailable specification

*Figure 8-4* depicts the specification of the situation SituationCaregiverWithinRange, in which a patient and a caregiver are within 100 meters distance from each other. This situation is a specialization of SituationWithinRange presented in *Figure 7-7*.

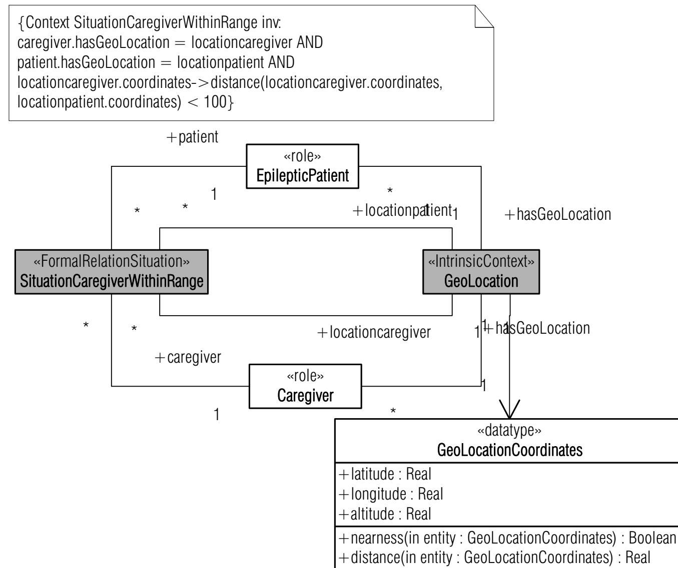



*Figure 8-4* SituationCaregiverWithin Range specification

## 8.1.2 Context information modelling

In section 8.1.1 we have presented conceptual context models without considering characteristics of the context information handled by the context-aware application, such as, e.g., the quality of the context information. As discussed in section 5.6, the information types exchanged between components in our context handling platform are *context information types*, as opposed to context types. When modelling context information types we consider whether context is sensed, derived, learned or provided, and we take quality of context into account.

*Figure 8-5* depict the context information types exchanged in the epileptic scenario.

«datatype»
GeoLocationMeasurement
+personID : String
+geoLocationCoordinates : GeoLocationCoordinates
+precision : RangePrecision
+freshness : Freshness
+origin : Origin
+probabilityOfCorrectness : Real

«datatype»
CaregiverStatusMeasurement
+caregiverID : String
+caregiverStatus : CaregiverStatusEnum
+freshness : Freshness
+origin : Origin
+probabilityOfCorrectness : Real

«datatype»
HazardousActivityMeasurement
+patientID : String
+hazardousActivityValue : Boolean
+freshness : Freshness
+origin : Origin
+probabilityOfCorrectness : Real

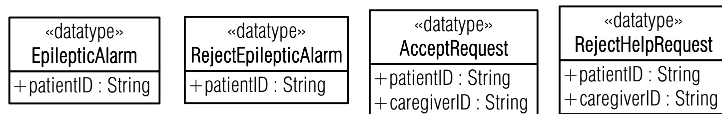
*Figure 8-5* Healthcare application measurement datatypes

These measurement datatypes refer to serialized values of the respective context types. For example, the GeoLocationMeasurement datatype refers to the person's unique identifier (attribute personID), and his/her measured geographical location coordinates (attribute geoLocationCoordinates of type GeoLocationCoordinates). In addition, various quality of context attributes are used, as discussed in Chapter 5. For the CaregiverStatusMeasurement datatype we define two attributes, which indicate the unique identifier of the caregiver

(caregiverID) and his/her availability status (caregiverStatus). Similarly, the HazardousActivityMeasurement datatype defines two attributes, which indicate the patient identifier (patientID) and whether he/she is undergoing a possibility hazardous activity (hazardousActivityValue).

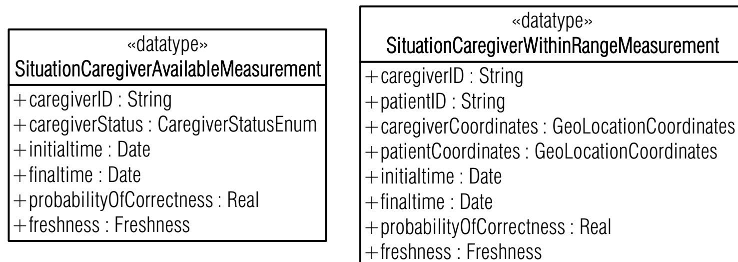
The primitive events supported are EpilepticAlarm (EpilepticPatient), RejectEpilepticAlarm (EpilepticPatient), AcceptHelpRequest (EpilepticPatient, Caregiver), and RejectHelpRequest (EpilepticPatient, Caregiver). *Figure 8-6* depicts the event notification structures, which aim at notifying requesters of occurrences of these events. Possible parameters for event notifications are specified as datatype attributes.

*Figure 8-6* Primitive event notification datatypes




With respect to situation types, two measurement types are necessary, namely SituationCaregiverAvailableMeasurement, and SituationCaregiverWithinRangeMeasurement. These measurement datatypes refer to serialized values of situation instances, i.e. the unique identifiers of entities, the context values as datatype values (e.g., GeoLocationCoordinates datatype), and the situation's initial and final times. *Figure 8-7* depicts these measurements datatypes.

*Figure 8-7* Situation datatype measurements




### 8.1.3 Healthcare application structural design

*Figure 8-8* depicts an overview of the healthcare context-aware application using the formal graphical representation adopted by this thesis, which is discussed in Chapter 2, section 2.2.1. This particular style for structuring context-aware applications and platforms has been extensively discussed in Chapters 2 and 4. Interactions taking place in interaction points ip-$a_i$ and ip-$b_i$ model the activities performed in cooperation between the users and the healthcare context-aware application, and between the users' contexts and the healthcare context-aware application, respectively. All possible information types that may be established in both ip-a and ip-b are defined in a *context information model*, which has been presented in section 8.1.2.

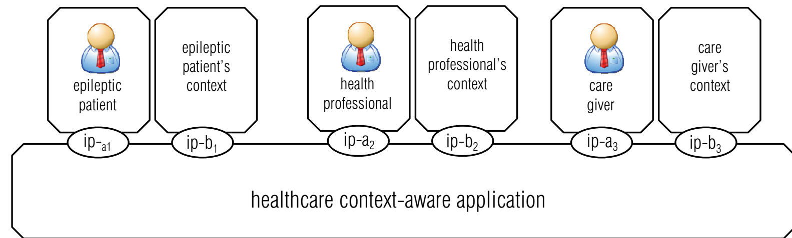



*Figure 8-8* Healthcare application general view

In order to cope with complexity and cost-effectiveness, the healthcare context-aware application is developed using the services offered by our context handling platform, which has been discussed in general in Chapter 4. *Figure 8-9* illustrates the healthcare application developed with the support of our context handling platform. The healthcare context-aware application uses generic services offered by the platform (grey area in *Figure 8-9*) and also implements specific services, which are offered by *application-specific components*. Application-specific services typically consist of application-specific functions that are not worth generalizing, since they are bound to the purpose of the application. These services should not be included in the context handling platform. Some context information values are often privacy sensitive, and therefore should not be shared.

In general, functions implementing user interfaces, which capture and render results from/to users, should be implemented as application-specific components.

The patient's application specific components should also include the functions to detect the upcoming epileptic seizure, which are too specific to be generalized in the platform. In addition, the patient's application-specific components should be able to determine whether the patient is currently undergoing a hazardous activity or not.

The health professional's application-specific components should include the functionality to filter and render biosignal streaming information. The caregiver's application-specific components should include the functionality to capture the status of the caregiver, and to capture the caregiver's acceptance or rejection for a particular request for help.

All the application-specific components are developed by the application developers. These components run on the mobile device carried by their respective users. Therefore, for each particular user there are particular instances of application-specific components, which are running on that user's mobile device.

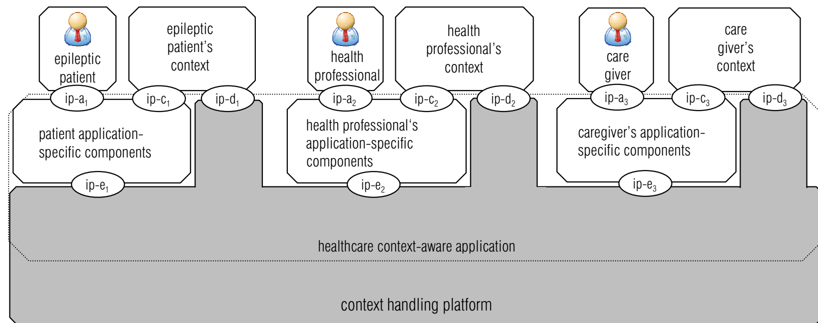



*Figure 8-9* Context handling platform offering support to the healthcare context-aware application

In *Figure 8-9*, users interact with application-specific components through interaction points of type ip-a. Interaction points of type ip-c and ip-d enable the users' contexts to interact with application-specific components and with the platform, respectively. Interaction points of type ip-c represent the mechanisms that are used to capture context, which are application-specific and cannot be shared, such as the detection of an upcoming epileptic seizure. Analogously, interaction points of type ip-d represent the mechanisms that are used to capture context information and may be reused by various other applications via the platform. For example, the mechanism that is used to capture geographical location information can be shared among various context-aware applications, and is, therefore, provided by the context handling platform. Interaction points of type ip-e enable interactions between application-specific components and the platform. This allows, for example, application-specific services to make use of shared context information.

### 8.1.4 Context provisioning services

So far, we have (i) modelled the application's universe of discourse and state-of-affairs by specifying relevant context and situation types, respectively; (ii) identified context and situation information measurement datatypes; (iii) refined the healthcare application into application-specific parts and generic parts, and (iv) distinguished the specific and generic functionality that should be realized in the specific-application parts and in the shared platform, respectively. We should now be able to identify the context processor components (context sources and managers), which are needed for capturing the relevant context and situation information types.

The application components that are capable of offering epileptic alarms, patient's activity information, caregiver status information and caregiver's acceptance or rejection for help notifications, play the role of *context sources*, as discussed in Chapter 4. The services offered by these components should be registered with the discovery service so that the platform is capable of finding them when necessary. Geographical location

information can be provided by the context handling platform, which is capable of offering geographical location information based on GSM cell triangulation.

We also need context manager components capable of detecting the particular situation types identified before. The services offered by these context manager components should also be registered with the discovery service so that the platform is capable of finding them when necessary.

Given these context information requirements, and the structural configuration of the healthcare application, we conclude that the following context sources should be developed: EpilepticAlarmContextSource ($CS_1$), GeoLocationContextSource ($CS_2$), HazardousActivityContextSource ($CS_3$), CaregiverStatusContextSource ($CS_4$), and the AcceptRequestContextSource ($CS_5$). Except for the GeoLocationContextSource ($CS_2$), all the other context sources are specific, and should be realized as part of the application-specific parts.

Given the situation information requirements, we conclude that the following context managers should be developed: SituationWithinRangeContextManagers ($CM_1$), and SituationAvailableContextManager ($CM_2$). These context manager components can be generalized, and therefore, are part of the context handling platform.

*Figure 8-10* depicts the context sources and managers as part of application-specific components, and part of the context-handling platform.

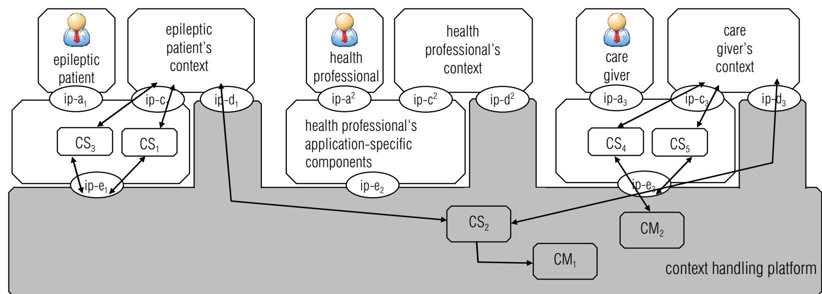



*Figure 8-10* Context sources in the application-specific components and in the platform

Several realization alternatives to implement the context sources are possible. Application developers are free to choose the most preferable alternative for their own purpose. An example of possible realization alternative for the EpilepticAlarmContextSource ($CS_1$) implements the algorithm for the detection of upcoming epileptic seizure defined by Roessingh research and development [110]. $CS_1$ interacts with the patient's context through interaction points of type ip-$c_1$ in order to gather the patient's biosignals. A possible implementation of the GeoLocationContextSource ($CS_2$) calculates the users' geographical locations using a GSM cell-based location mechanism, and is implemented by the platform developers. $CS_2$ interacts with the

patient's and the caregiver's contexts through interaction points of type ip-d in order to obtain signalling information from their mobile devices.

A possible implementation of the HazardousActivityContextSource ($CS_3$) takes an explicit input provided by the patient, when he/she undergoes a possible hazardous activity. As we have discussed in Chapter 2, in our definition of context information we do not distinguish between manually provided and automatically acquired (sensed) context information. Although in this application some context information values are manually provided by the user, these values still represent conditions in the user's context. Therefore, these values should be gathered through interaction points of type ip-$c_i$. Similarly, the CaregiveStatusContextSource ($CS_4$), and the AcceptRequestContextSource ($CS_5$) can be realized by means of explicit inputs provided by the caregiver, who provides his/her current status information and whether he/she accepts or not the request for help, respectively.

Both context manager components realize the situation detection mechanism we have discussed in Chapter 6. The SituationWithinRangeContextManager ($CM_1$) gathers location information from the GeoLocationContextSource ($CS_2$) in order to detect whether patients are nearby caregivers or not. The SituationAvailableContextManager ($CM_2$) gathers status information from the CaregiveStatusContextSource ($CS_4$) in order to reason about whether the caregiver is available or not.

*Figure 8-11* depicts the context source interfaces, which should be offered by the context sources just defined. The GeoLocationContextSource provides location information measurement notifications from time to time, e.g., on every minute. The CaregiverStatusContextSouce, and the HazardousActivityContextSource, in constrast, provide notifications of caregiver status measurements and hazardous activity measurements whenever these values are changed by the caregiver and the patient, respectively.

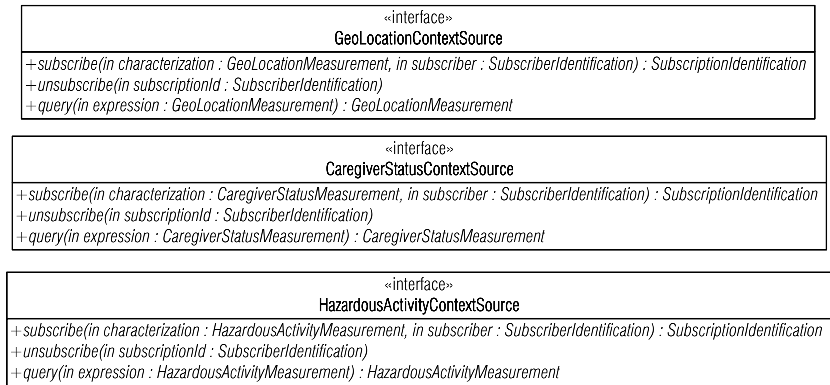



*Figure 8-11* Context source interfaces expected to offer the healthcare context measurement datatypes

*Figure 8-12* depicts context source interfaces that provide primitive event notifications, i.e. the interfaces of the context sources EpilepticAlarmContextSource and AcceptRequestContextSource.

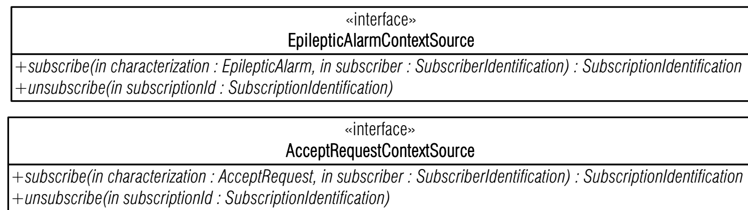



*Figure 8-12* Context source interfaces expected to offer the healthcare primitive event notification datatypes

*Figure 8-13* depicts the context manager interfaces for the context managers SituationAvailableContextManager and the SituationWithinRangeContextManager.

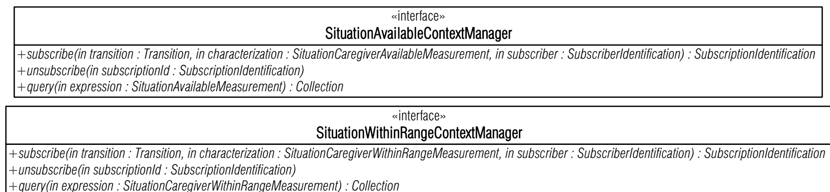



*Figure 8-13* Healthcare application situation measurement datatype and the respective context manager component

### 8.1.5 Action services

The application-specific components running on the users devices should implement action services in order to receive the invocations from the context handling platform. The invocations, in turn, are responses to particular occurrences of events and condition evaluations. We have discussed action services in detail throughout Chapters 2, 4 and 7. Three types of action services should be offered, as follows:

- PatientActionService ($AS_1$): action service offered by a patient's application-specific component. It implements the indicateEpilepticSeizure operation, which receives the notification of a possible epileptic seizure;

– ProfessionalActionService ($AS_2$): action service offered by a health professional's application-specific component. It implements logSeizure operation, which receives the notification of an epileptic alarm and logs it;
– CaregiverActionService ($AS_3$): action service offered by a caregiver's application-specific component. It implements the requestHelp and indicateAcceptance operations, which receives the requests for helping a particular patient, and notifies the acceptance of a certain caregiver, respectively;

*Figure 8-14* depicts the specification of the interfaces offered by these action service components.

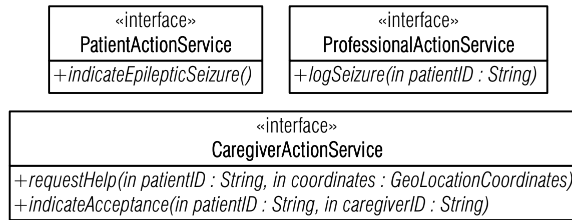



*Figure 8-14* Specification of action services to be offered by the application-specific components

### 8.1.6 Controlling services

In order to delegate required pieces of reactive behaviour to the platform, we should specify ECA-DL rules that represent these behaviours. The controlling services and the ECA-DL language have been extensively discussed in Chapter 7. The following reactive behaviours are required:

1. *SeizureAlarm*: upon an epileptic seizure alarm, the application running on the patient's device should be notified of the possibility of an epileptic seizure and that the patient is currently performing a hazardous activity. This patient's application may launch a voice message warning the patient to stop such activity in the next minutes;
2. *HelpRequestNotification*: upon an epileptic seizure alarm, a collection of caregivers should be notified that a patient is in need of help. Only caregivers that are available and nearby the patient should receive the notification. Upon receiving such notification, the application running on the caregiver's device may plot on a map a possible route between the caregiver and the patient;
3. *HelpAcceptedNotificafion*: upon receiving an acceptance from a particular caregiver, the caregivers that had previously received the notification for help should be notified that one of the caregivers has already accepted to help that particular patient;
4. *LogSeizureAlarm*: upon an epileptic seizure alarm, log the alarm on the health professional's application for further analysis.

***Healthcare action resolver component***

An action resolver component receives invocations from a controller component aiming at dispatching a number of actions. We have defined a *healthcare action resolver* that implements all possible actions that can be referred to by ECA-DL rules running in the controller component. When events occur and conditions are matched, the controller invokes actions on the healthcare action resolver, which may in turn dispatch the appropriate invocations to the right *action services*, which are implemented in the respective application-specific components. A single invocation of an action implemented in the resolver may generate several invocations on the action services running on the users' application-specific components.

The following actions should be implemented by the healthcare action resolver component in order to realize the aforementioned behaviours:

– notifyPatientApplication is required to realize reactive behaviour *SeizureAlarm*, which aims at notifying the proper patient's application of his/her possible seizure alarm. This action receives as argument the patient's unique identifier, with which it is possible to acquire the address of the application's service endpoint on the patient's device. When the patient's address is acquired, the indicateEpilepticSeizure operation offered by the PatientActionService is invoked;
– notifyCaregiversApplications is required to realize reactive behaviour *HelpRequestNotification*, which aims at notifying the proper caregivers' applications. The arguments accepted by this action are (i) the location of the patient; and (ii) the collection of caregivers who need to be notified. For each of these caregivers, the requestHelp operation offered by the CaregiverActionService is invoked;
– notifyAcceptanceCaregivers is required to realize reactive behaviour *HelpAcceptedNotification*, which aims at notifying the proper caregivers' applications. This action receives as arguments (i) the collection of caregivers who need to be notified; (ii) the patient that needs help; and (iii) the caregiver that accepted to help. For each of the caregivers who need to be notified, the indicateAcceptance operation offered by the CaregiverActionService is invoked;
– logEpilepticAlarm is required to realize reactive behaviour *LogSeizureAlarm*, which aims at notifying the health professionals' applications that an epileptic alarm has occurred. This action receives as argument (i) the unique identifier of the patient having the seizure; and (ii) a collection with the health professionals' identifiers. The logSeizure operation offered by the ProfessinalActionService is invoked.

*Figure 8-15* depicts the specification of the *healthcare action resolver* interface that dispatches the required action invocations to the proper implementations. These implementations are distributed and running on the users' devices.

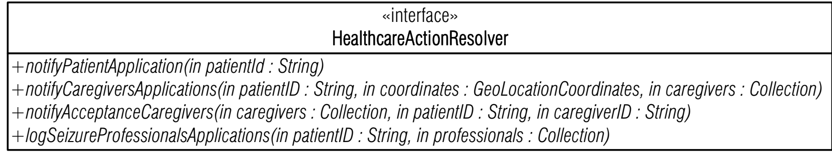



*Figure 8-15* Specification of action resolver services for the Healthcare application

### *Component configuration*

*Figure 8-16* depicts the complete configuration of the components that are necessary to realize the healthcare context-aware application. It refines the architecture presented in *Figure 8-8*, incorporating the *healthcare controller* component, the *healthcare action resolver* component, and the *application-specific action service* components. Such types of refinements have been extensively discussed in Chapter 4. The gray area indicates the context handling platform part. As already discussed, the application-specific parts consist of specific functions that should not be generalized in the platform. These include several context sources and the application-specific action services. Context manager components and the location context source are implemented as part of the platform.

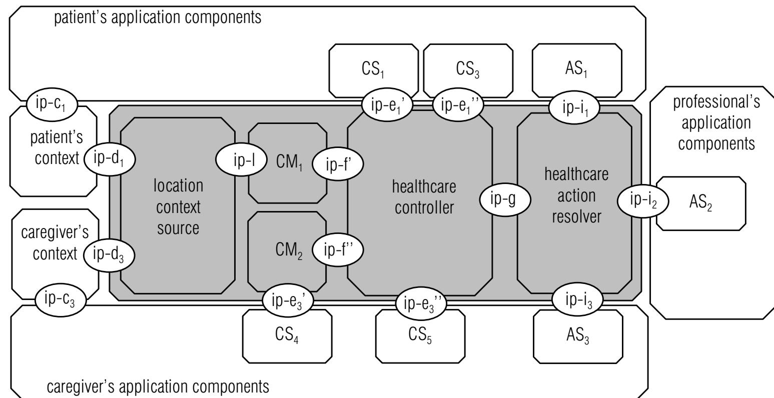



*Figure 8-16* Configuration of components for the healthcare application

We have proposed a single controller component to realize the reactive behaviours of the healthcare application. However, it would be also possible to use several instances of the controller component, each containing specific ECA-DL rules. Since ECA-DL rules are independent from each other, they can be distributed arbitrarily over different instances of controller components.

### *ECA-DL rules*

Each of the ECA-DL rules discussed below aims at specifying a particular behaviour required by the healthcare application. These rules are offered to the healthcare controller component at platform runtime, i.e. at the

controller component operation time. We assume for the healthcare application scenario that the ECA-DL rules presented here are defined at application design-time, and at platform runtime. However, it is also possible that ECA-DL rules are defined at application runtime. For example, the health professional could be provided with a user interface that allowed him/her to define desired reactive behaviours, which could be translated to ECA-DL by some specific component running on the professional's device. In this case, ECA-DL rules would be provided to the controller component by the health professional's application-specific components. We have discussed these alternative configurations in Chapter 7. In addition, other rules, different from the ones presented here, could also be specified. The only requirement is that the context and situation values, events, and actions referred to by the rules should be available to the controller component. The Jess rules derived from these ECA-DL rules can be found in Appendix D.

The following ECA-DL rule (ECARule1) aims at realizing the reactive behaviour *SeizureAlarm*. It uses a scope clause to define that this rule should be applied to all epileptic patients registered in the controller component. The When clause checks whether this patient is currently performing any potentially hazardous activity. When both the event happens and the condition in the when clause is true, the action notifyPatientApplication is invoked.

```
Scope (EpilepticPatient.*; p)
{
    Upon EpilepticAlarm (p)
    When p.hasHazardousActivity.hazardousvalue
    Do NotifyPatientApplication (p)
}
```

The following ECA-DL rule (ECARule2) aims at realizing the reactive behaviour *HelpRequestNotification*. This rule uses the same scope clause as the rule above. In addition, there is no when clause in the rule. Upon receiving event notification of an epileptic alarm for any epileptic patient, the action NotifyCaregiversApplications is invoked. The select clause selects all the patient's caregivers who are within range and available. The result collection of this select is passed as one of the arguments of the action.

```
Scope (EpilepticPatient.*; p)
{
  Upon EpilepticAlarm (p)
  Do NotifyCaregiversApplications ( p, p.hasGeoLocation.coordinates,
       Select (Caregiver.*; care; isCaregiverOf (care, p) and SituationWithinRange (p, care)
```

```
                and SituationCaregiverAvailable (care)))
}
```

The following ECA-DL rule (ECARule3) aims at realizing the reactive behaviour *HelpAcceptedNotification*. This rule uses the same scope clause as the previous rules. The upon clause specifies an event composition, in which Ev1 (EpilepticAlarm (p)) should occur followed by Ev2 (AcceptRequest (p)). This means that this composite event happens every time there is an epileptic seizure alarm, followed by an acceptance for helping the patient having the seizure. As shown in *Figure 8-6*, the AcceptRequest accepts two arguments, namely a caregiver and a patient. In this example, only a patient is provided as argument. This way, we can filter the AcceptRequest event notifications for patient p, from any caregiver. Upon the occurrence of this composite event, the action notifyAcceptanceCaregivers is invoked. The select clause selects all the patient's caregivers who are within range, and are different from the caregiver who accepted the request. The identification of the caregiver who accepted the request is obtained by accessing one of the attributes of the event, namely Ev2.caregiverID. The result collection of this select is passed as one of the arguments for the action.

```
Scope (EpilepticPatient.*; p)
{
  Upon Ev1: EpilepticAlarm (p); Ev2: AcceptHelpRequest (p)
  Do notifyAcceptanceCaregivers (
        Select (CareGiver.*; care; isCareGiverOf (care, p) and
                    SituationWithinRange (care, p) and
                    care <> Ev2.caregiverID), p, Ev2.caregiverID)
}
```

The following ECA-DL rule (ECARule4) aims at realizing the reactive behaviour *LogSeizureAlarm*. Upon receiving an epileptic alarm notification for any epileptic patient, the action logEpilepticAlarm is invoked. The select clause selects all the patient's health professionals.

```
Scope (EpilepticPatient.*; p)
{
  Upon EpilepticAlarm (p)
  Do logEpilepticAlarm (p, Select (HealthProfessional.*; prof; isHealthProfessionalOf (prof, p)))
}
```

### 8.1.7 Deployment

The situation detection specification should be deployed in the context managers, and the ECA-DL rules in the controller component. Deploying ECA-DL rules consists of submitting a textual representation of the rule with the controller component. This has been discussed in Chapter 7. For the situation specification, only Jess rules and Java code can be deployed in a context manager component. The systematic derivation of Jess rules and Java code from UML and OCL, respectively, has been extensively discussed in Chapter 6. Appendix D shows the Jess rules that correspond to the situation specification and ECA-DL rules.

## 8.2 The Context-Aware Policy Management Application

Suppose we would like to extend the healthcare scenario presented in section 8.1 in order to allow caregivers and health professionals to access, for example, Mr. Janssen's private information in particular situations. We may define, for privacy reasons, that the health professional may access Mr. Janssen's biosignals only for 15 minutes after an epileptic alarm notification. Similarly, we could also define that patients can only check his/her caregiver's availability status upon an epileptic alarm prediction. These types of privacy rules are commonly referred to as *policies*. In general, managing policies in a context-aware application concerns controlling access to context information, enforcing user's privacy, and managing trust relationships among end-users and components.

Context-aware applications and platforms may potentially handle thousands of entities (e.g., end-users, service providers, context providers), making the management of policies difficult [84]. In such cases, a policy management tool that assists and automates management of policies is desirable. Standard policy management tools, however, typically do not offer support for context-aware policy management, which allow the definition of policies based on the users' context information. These tools usually offer support for the definition of static policies, in which the entities for which these policies apply are known beforehand.

In contrast, context-aware applications and platforms require dynamic policy management, in which policies may be created and destroyed when entities enter or leave particular context and situation conditions. In order to tackle this issue, we propose here a dynamic policy management mechanism that defines policies based on the users' context information. This mechanism is realized by means of our situation detection framework and the controlling services. For this particular scenario, we reuse the

context and situation models already presented in section 8.1.1, and the context and situation information models presented in section 8.1.2.

### 8.2.1 Policy management application structural design

*Figure 8-17* shows how the context handling platform offers support to the policy management application. This application uses generic services offered by the platform and also implements specific services (application-specific components). *Figure 8-17* extends *Figure 8-9* by including a policy administrator user and his/her respective administrator application-specific components. By means of interaction points of type ip-$a_4$, the policy administrator defines the policies of interest to this application, in addition to configuration information such as starting commands. The policy administrator-specific components map the behaviours gathered from the policy administrator to ECA-DL rules, which are offered to the controller platform through interaction points of ip-$e_4$. In addition, interaction points of type ip-$e_4$ enable the administrator application to gather management information regarding the policy enforcement activities from the platform.

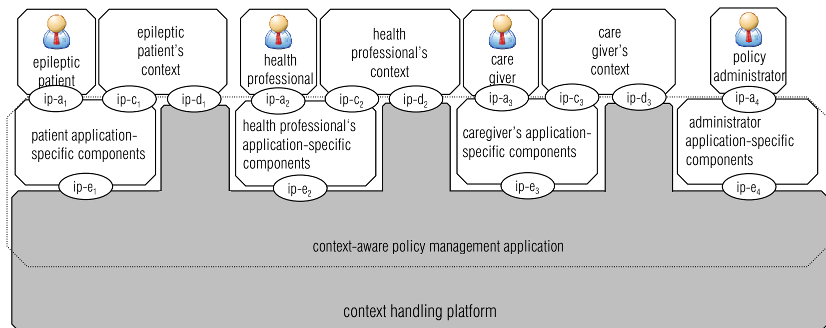



*Figure 8-17* Overview of the context-aware policy management application structural design

### 8.2.2 Policy enforcement components

In the policy management application, the context source components also implement *policy enforcement interfaces*. These interfaces allow context sources to be configured in order to fulfil particular policy requirements, which are context and situation-dependent. For example, the EpilepticAlarmContextSource should permit configuration of policies, such as the one to only allow caregivers and health professionals that are currently in particular context situations to access the patient's vital signs. Similarly, it should be possible to configure the CaregiverStatusContextSource to only allow patients and health professionals that are in particular context situations to access the caregivers' status information.

*Figure 8-18* depicts the policy enforcement interfaces that should be implemented by the EpilepticAlarmContextSource ($CS_1$), and by the CaregiverStatusContextSource ($CS_4$).

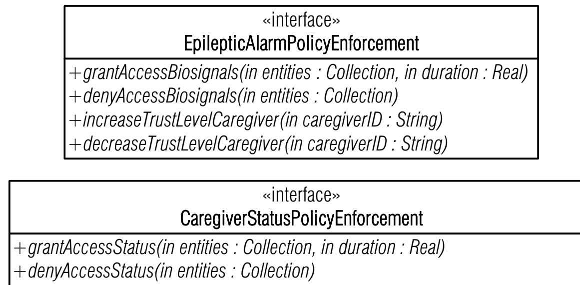



*Figure 8-18* Policy enforcement interfaces to be implemented by context source components

The operation grantAccessBiosignals grants access to the patient's biosignals information to a collection of entities (caregivers and health professionals), which is provided as argument to this operation. The denyAccessBiosignals operation configures the EpilepticAlarmContextSource to deny access to the patient's biosignals to a particular collection of entities, which is passed as argument to this operation.

A patient's application maintains *trust values* that are assigned to the patient's caregivers. The aim of a trust value is to give the patient an indication of how much she/he can rely on a particular caregiver. The trust value changes as a result of the caregiver's reaction to a request. When a caregiver accepts a request for help (AcceptRequest event notification), his trust value should be increased, and when a caregiver rejects to help (RejectHelpRequest event notification), his trust value should be decreased. The operation increaseTrustLevelCaregiver allows the patient's application to increase the trust value of a particular caregiver, and the operation decreaseTrustLevelCaregiver allows the patient's application to decrease the trust value of particular caregiver.

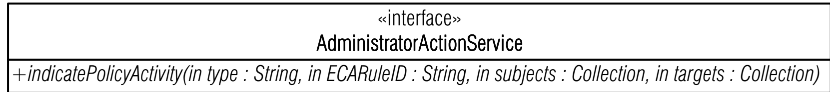
With respect to the CaregiverStatusPolicyEnforcement interface, we defined two operations, namely grantAccessStatus, and denyAccessStatus. The grantAccessStatus operation grants access to status information of this particular caregiver to collection of entities, for a certain period of time. Similarly, the denyAccessStatus operation denies access to the caregiver status information to a given collection of entities.

### 8.2.3 Policy administrator action service

The administrator's application-specific components implement management functionality that allows the administrator to have a global view of the policies that are applied and to which entities they refer. Therefore, the administrator's application-specific components should be informed of all policies that are applied in the platform. In order to realize that, we define an AdministratorActionService ($AS_4$) component that receives invocations from the context handling platform. This action service component offers operations to inform the administrator's application-

specific components of policy related activities. *Figure 8-19* depicts the interface implemented by the AdministratorActionService component.

*Figure 8-19* Administrator action service interface




The operation indicatePolicyActivity informs the type of the policy (e.g, "access control of patient's biosignals"), the identification of the ECA-DL rule that triggers the policy, the *subjects* and the *targets* referred to in this particular policy. Subjects and targets are entities that play different roles depending on the type of policy. For example, in an access control policy, a subject is the entity being granted or denied with access to information about a target. In the example of access control to the patient's biosignals, the patient is the target and the caregiver is the subject. In trust management policies, the subjects refer to the entities that maintain trust values about targets. In our example, a patient that maintains trust values about caregivers is the subject, while the caregivers are the targets.

### 8.2.4 Controlling services

In order to delegate required pieces of reactive behaviour to the platform, ECA-DL rules that represent these behaviours should be defined first. The following reactive behaviours are required by the policy management application:

1. *GrantAccess*: upon a patient's epileptic seizure alarm, the patient's caregivers that are available and nearby should be granted with rights to access the patient's biosignals. In addition, the status information of caregivers who are nearby should become available for the patient for 30 minutes;
2. *DenyAccess*: upon a caregiver's acceptance notification, the other caregivers that had previously received the request for help should be denied rights to access the patient's biosignals;
3. *IncreaseTrust*: upon a caregiver's acceptance notification, the patient's application should increase his/her trust value on the caregiver that accepted the request to help;
4. *DecreaseTrust*: upon a caregiver's rejection notification, the patient's application should decrease his/her trust value on the caregiver that rejected a request for help.

For management reasons, the administrator should have a global view of policy management activities that are occurring within the platform. In order to inform the administrator's application-specific components of such activities, whenever a policy is deployed or removed, the policy administrator action service should be invoked.

***Policy management action resolver component***

In contrast to the previous application in which the healthcare action resolver component only invokes action services on application-specific parts, in the policy management application, context sources ($CS_3$ and $CS_4$) are invoked as the result of ECA-DL rules execution. The following actions are implemented by the policy management action resolver in order to realize the aforementioned behaviours:

- grantAccessControlPolicyPatient: used to realize reactive behaviour *GrantAccess* in order to deploy access control policies over the patient's biosignals. It receives as arguments (i) the patient to be configured; and (ii) the collection of entities for which biosignals should be granted to. It invokes the operation grantAccessBiosignals on the EpilepticAlarmContextSource running on the patient's device;
- denyAccessControlPolicyPatient: used in reactive behaviour *DenyAccess* to remove access control policies over the patient's biosignals. It receives as arguments (i) the patient to be configured; and (ii) the collection of entities for which biosignals should be denied to. It invokes the operation denyAcccessBiosignals on the EpilepticAlarmContextSource running on the patient's device;
- grantPolicyCaregiversStatus : used to realize reactive behaviour *GrantAccess* in order to deploy access control policies over the caregiver's status information. It receives as arguments (i) a collection of caregivers to be configured; and (ii) the patient to which the status information should be granted; For each caregiver, it invokes the operation grantAccessStatus on the CaregiverStatusAvailabilityContextSource running on the caregivers' devices;
- increaseTrustPolicyPatient: used to realize reactive behaviour *IncreaseTrust* in order to deploy trust management policies over the caregiver's trust values. It receives as arguments (i) a patient to be configured; and (ii) the caregiver for which the trust level should be increased. It invokes the operation increaseTrustLevelCaregiver on the EpilepticAlarmContextSource running on the patient's device;
- decreaseTrustPolicyPatient: used to realize reactive behaviour *DecreaseTrust* in order to deploy trust management policies over the caregiver's trust values. It receives as arguments (i) a patient to be configured; and (ii) the caregiver for which the trust level should be decreased. It invokes the operation decreaseTrustLevelCaregiver on the EpilepticAlarmContextSource running on the patient's device.

Each of these actions invokes the indicatePolicyActivity on the action service running on the administrator device in order to inform the administrator of the particular policy activity that is currently taking place. *Figure 8-20* depicts the specification of the action resolver component interface for the policy management application.

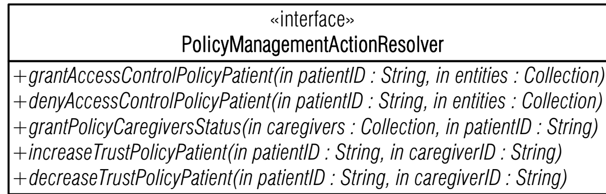



*Figure 8-20* Specification of action resolver interface for the policy management application

### *Configuration of components*

*Figure 8-21* depicts the complete configuration of the components that are necessary to realize the policy management application. It refines the architecture presented in *Figure 8-17*, incorporating the *policy management controller* component, the *policy management action resolver* component, and the *application-specific action service* components. Differently from the healthcare application, the context sources also implement policy enforcement points, which are invoked by the action resolver component through interaction points of type ip-$i_1$ and ip-$i_3$. Invocations to the administrator's application are performed by means of interaction points of type ip-$i_2$. Context source $CS_5$ has been omitted in *Figure 8-21* for clarity.

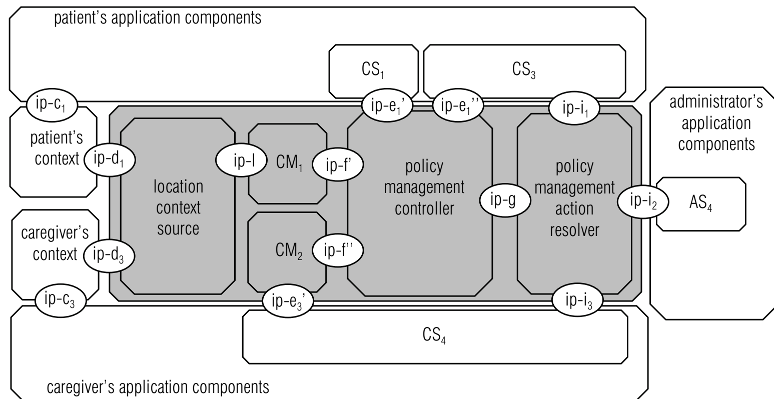



*Figure 8-21* Component configuration for the policy management application

### *ECA-DL*

Each of the following ECA-DL rules aims at specifying a particular behaviour required by the policy management application. These rules are offered to the policy management controller by the administrator's application at platform runtime. The interaction point that enables the administrator's application to provide ECA-DL rules to the platform is not shown in *Figure 8-21* for the sake of clarity. The Jess rules derived from these ECA-DL rules can be found in Appendix D.

The following ECA-DL rule (ECARule5) aims at realizing the reactive behaviour *GrantAccess*. The scope and upon clauses are similar to the healthcare application ECA-DL rules, which have been discussed in section

8.1.6. Upon receiving event notification of an epileptic alarm for any epileptic patient, actions grantAccessControlPolicyPatient and grantPolicyCaregiversStatus are invoked.

```
Scope (EpilepticPatient.*; p)
{
  Upon EpilepticAlarm (p)
  Do grantAccessControlPolicyPatient (p,
        Select (Caregiver.*; care; isCaregiverOf (care, p) and SituationWithinRange (p, care)
              and SituationCaregiverAvailable (care)));
     grantPolicyCaregiversStatus (Select (Caregiver.*; care; isCaregiverOf (care, p) and
                 SituationWithinRange (p, care) and SituationCaregiverAvailable (care)), p)
}
```

The following ECA-DL rule (ECARule6) aims at realizing the reactive behaviour *DenyAccess*. The upon clause defines that an epileptic seizure alarm should be followed by a caregiver acceptance for helping the patient having the seizure. Upon the occurrence of this composite event, the action denyAccessControlPolicyPatient is invoked. This operation denies access control to all caregivers that were previously granted access to the patient's biosignals, except for the caregiver who accepted the request for helping.

```
Scope (EpilepticPatient.*; p)
{
  Upon Ev1: EpilepticAlarm (p); Ev2: AcceptHelpRequest (p)
  Do   denyAccessControlPolicyPatient (p,
             Select (CareGiver.*; care; isCareGiverOf (care, p) and
                   SituationWithinRange (care, p) and SituationCaregiverAvailable (care)
                   and care <> Ev2.caregiverID), p, Ev2.caregiverID));
}
```

The following ECA-DL rule (ECARule7) aims at realizing the reactive behaviour *IncreaseTrust*. The upon clause defines the same composite event as ECARule6. Upon the occurrence of this composite event, the action increaseTrustPolicyPatient is invoked, which increases the trust value of a particular caregiver.

```
Scope (EpilepticPatient.*; p)
{
  Upon Ev1: EpilepticAlarm (p); Ev2: AcceptHelpRequest (p)
  Do   increaseTrustPolicyPatient (p, Ev2.caregiverID)
}
```

The following ECA-DL rule (ECARule8) aims at realizing the reactive behaviour *DecreaseTrust*. The scope clause applies the rule to all the caregivers in the system. Upon receiving a RejectHelpRequest event notification, the action decreaseTrustPolicyPatient is invoked. The identification of the patient is retrieved as one of the parameters of the RejectHelpRequest event notification.

```
Scope (Caregiver.*; c)
{
  Upon Ev:RejectHelpRequest (c)
  Do decreaseTrustPolicyPatient (Ev.patientID, c)
}
```

## 8.3 The Healthcare Application Prototype

We have built a prototype that implements the healthcare application. The objective of this prototype is twofold:

- To demonstrate the *feasibility* of our approach. The prototype serves to demonstrate that the abstractions proposed in this thesis (context and situation modelling), and the context handling platform can be built in a computing system; and
- To allow the assessment of the performance and the scalability of the approach proposed.

### 8.3.1 System configuration

The prototype implements a *design product*, which is the result of our design efforts presented in section 8.1. The design product includes (i) the Java code that implements the context and situation model; (ii) the Jess rules for situation detection; (iii) the Jess rules for ECA-DL rules; (iv) the architecture presented in *Figure 8-16*; (v) the context information measurement datatypes; and (vi) the specification of the component interfaces.

We have implemented the components using Java as the programming language, Java RMI [63] for enabling the distribution of the components, and Eclipse [34] as the development environment. We have implemented in the prototype all the components that were identified in our design efforts, namely the context sources, the context managers, the controller, the action resolver, and the action services (see architecture in *Figure 8-16*). Application-specific components that are neither context sources, nor action services have not been implemented, i.e. we have not implemented end-user interfaces, and other user-specific functionality.

***Context sources***

We have simulated the production of context information and events. The GeoLocationContextSource randomly generates geographical location coordinates for its users, but respecting a maximum variation of the location changes. The EpilepticAlarmContextSource and AcceptRequestContextSource also randomly generate EpilepticAlarm and AcceptRequest events, respectively. Similarly, the CaregiverStatusContextSource and the HazardousActivityContextSource randomly changes the availability status of caregivers and whether the user is performing a potentially hazardous activity or not. The context sources do not implement the context models proposed, since they generate simple isolated values, which are not yet combined with other context values. We assume, however, that users in the system (caregivers, patients and health professionals) are uniquely identified by an id, which is shared among the components.

***Context managers***

The context managers to detect situations SituationCaregiverWithinRange, and SituationCaregiverAvailable have been implemented as discussed in Chapter 6. The SituationCaregiverWithinRangeContextManager gathers context information from the GeoLocationContextSource through a callback interface, which allows the GeoLocationContextSource to push location information to the SituationCaregiverWithinRangeContextManager component whenever a new location value is generated. When location information is received from the location context source, the SituationCaregiverWithinRangeContextManager updates local Java objects, which are automatically shadowed in the Jess working memory. Location values are exchanged using the GeoLocationMeasurement datatype depicted in *Figure 8-5*.

The SituationCaregiverWithinRangeContextManager detects when patients and caregivers are nearby each other by means of Jess rules running on the local Jess engine. In order to detect this type of situation, this context manager implements part of the situation model, in accordance to the specification of the SituationCareGiverWithinRange, depicted in *Figure 8-4*.

Similarly, the SituationCaregiverAvailableContextManager component implements the situation model presented in *Figure 8-3*. In order to reason about the availability of the caregivers, it gathers context information from the CaregiverStatusContextSource through a callback interface, which allows the CaregiverStatusContextSource to push caregiver status information to the SituationCaregiverAvailableContextManager whenever the status of a particular caregiver changes. Caregiver status information is exchanged following the CaregiverStatusMeasurement datatype depicted in *Figure 8-5*. The SituationCaregiverAvailableContextManager component runs a local Jess engine, which detects when the situation begins and ceases to hold.

### *Controller*

The healthcare controller component gathers context information from context sources and managers in order to gather event notifications and condition values. Since the controller integrates context and situation information, it implements the complete context and situation models, as presented in *Figure 8-2*, *Figure 8-3* and *Figure 8-4*. We have implemented callback interfaces in the controller, which allows the context sources and managers to push context and situation information in the controller. When these callback methods are invoked, context and situation information is received by the controller and included into the local Jess working memory, by means of shadow facts. ECA-DL rules are continuously-running in the local Jess engine.

The controller implements the event structures, which are necessary to realize event consumption and composition, as presented in Chapter 7. According to the ECA-DL rules defined for the healthcare application, two types of event notifications are expected, namely EpilepticAlarm and AcceptRequest. As we have defined in Chapter 7, all event notifications expected from context sources should be registered with the controller. Since the ECA-DL rules are scoped, there are EpilepticAlarm event registrations for each epileptic patient and for each rule that refers to this event. Suppose, for example, only Mr. Janssen is currently supported as an epileptic patient, and his caregivers are Mary and Alice. The controller maintains event registrations, as depicted in *Table 7-6*. In this way, whenever an epileptic alarm event notification is received, such as, for example EpilepticAlarm (Mr Janssen), the controller matches this notification with the registrations available (*Table 7-6*). Since there are four matches, the controller includes four instances of the EpilepticAlarm event in the working memory, one for each rule referring to that event. Similarly when event notifications of AcceptRequest (Mr.Janssen, caregiver) are received, the controller finds two matches, one for each caregiver, and creates two instances of this event in the working memory.

In total, our prototype runs three instances of the Jess engine, one for each context manager component, and one for the controller component. Context sources are not implemented using Jess engines.

*Table 8-11* Example of event registrations that are maintained in the controller component

| Event Description | ruleID | Parameters |
|---|---|---|
| EpilepticAlarm | ECARule1 | Mr. Janssen |
| EpilepticAlarm | ECARule2 | Mr. Janssen |
| EpilepticAlarm | ECARule3 | Mr. Janssen |
| EpilepticAlarm | ECARule4 | Mr. Janssen |
| AcceptRequest | ECARule3 | Mr. Janssen, Mary |
| AcceptRequest | ECARule3 | Mr. Janssen, Alice |

### 8.3.2 Indication of realization efforts

We argue in this thesis that our approach facilitates and simplifies application development since it provides a framework that systematically derives part of the application realization from its specification. This framework supports the derivation of the following application parts: (i) context models; (ii) situation detection; and (iii) reactive behaviours. In order to have an indication of how our approach facilitates and simplifies application development, we have attempted to quantify the specification and the realization efforts.

In order to provide a rough quantification of the specification efforts, we count the number of UML class diagrams, and the number of OCL and ECA-DL *lines of code (LOC)*. For roughly quantifying the realization efforts, we count the Java and the Jess lines of code that are systematically derived from the aforementioned specifications. For counting Java code we have used Metrics [35], an open-source Eclipse plug-in that gathers various metrics of Eclipse projects. *Table 8-12* shows these quantification values.

*Table 8-12* Comparison between specification and realization efforts

| UML classes (units) | OCL (LOC) | ECA-DL (LOC) | Java (LOC) | Jess (LOC) |
|---|---|---|---|---|
| 23 | 9 | 21 | 663 | 126 |

The results presented in *Table 8-12* indicate that the efforts required for the realization phase are a substantial part of the application development. Therefore, since our approach alleviates application realization, it facilitates and simplifies a substantial part of the application development.

### 8.3.3 Performance evaluation

In order to evaluate the scalability and the performance of the healthcare application, we have defined the *reaction time* evaluation parameter. Reaction time is defined as the period of time between generating an event and finally invoking an action of an ECA-DL rule that refers to that event. For example, we can assess the reaction time between an EpilepticAlarm event occurrence and triggering ECARule1, or we can assess the reaction time between an AcceptRequest event occurrence and triggering ECARule3. Therefore, the reaction time consists of the processing time that takes place between an event occurrence, and the invocation of the action. We capture the reaction time in the EpilepticAlarmContextSource, which receives a callback invocation from the controller when rules ECARule2 and ECARule3 are triggered. These callback invocations are included in the RHS (action part) of the Jess rules derived from ECARule2 and ECARule3, for the purpose of evaluating the performance. Therefore, every time one of these rules is executed, the controller's Jess engine invokes the EpilepticAlarmContextSource

component, which is then capable of calculating the reaction time. These callback invocations are not part of the normal application's behaviour.

***General configuration***

We have measured in our evaluation the reaction time between the occurrence of an EpilepticAlarm, and triggering ECARule2 and ECARule3. The reaction time between an occurrence of the EpilepticAlarm and triggering ECARule3 is bigger than between the same EpilepticAlarm occurrence and triggering ECARule2. This is due to the composition of events defined in ECARule3, which requires that an event AcceptRequest should occur after the EpilepticAlarm.

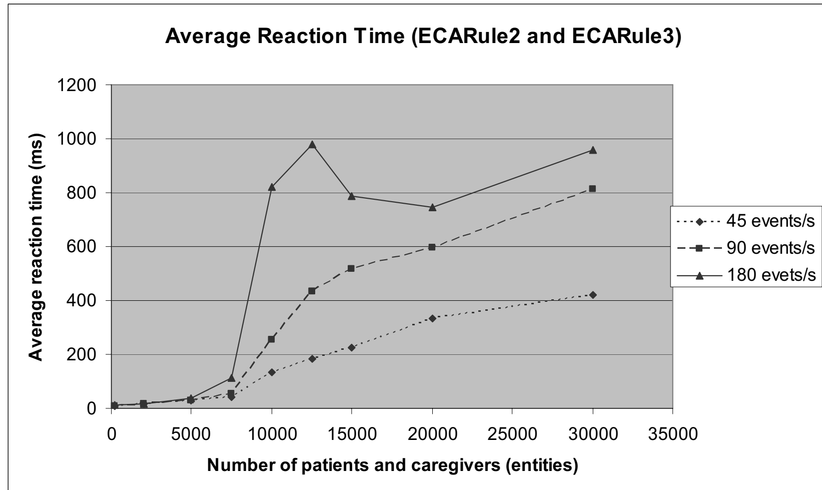
We have collected reaction time measurements for different numbers of patients and caregivers (entities), and for different amounts of events generated per second. For each pair (number of entities, number of events per second), we have collected 100 reaction time measurements, for which we calculate an *average reaction time* value. The average reaction time includes the reaction times for both ECARule2 and ECARule3.

We have performed tests in order to evaluate how the application behaves under extreme circumstances, such as with extreme numbers of users and events. These types of tests are often called *stress tests*. Stress testing aims at assessing the robustness and the availability of the system under heavy load. In order to perform such stress testing, we have observed and measured the reaction times when the system is loaded with up to 35000 entities, and is generating up to 450 events per second.

***Centralized configuration***

*Figure 8-22* depicts a graphic which demonstrates the increase of the average reaction time in milliseconds (Y axis) with respect to the number of entities (X axis). Three curves are shown, each one representing a particular number of events generated per second. For these measurements all the components are running on a single machine with 1GB of memory.

*Figure 8-22* Average reaction times for ECARule2 and ECARule3 in a centralized configuration




We have verified that about 85% of the reaction time consists of the processing time required by the Jess engine. The other 15% is due to the event consumption mechanism implemented by the controller component. The later could be improved by optimizing the event structures using, for example, *hash tables*.

The shape of these curves can be explained by considering how Jess works. Jess implements the Rete algorithm, which builds a network of nodes on memory, each node representing one or more patterns on the rule LHS. Facts that are being added to or removed from the working memory are processed by this network of nodes. At the bottom edges of the network are the nodes representing individual rules. When a set of facts filters all the way down to the edges of the network, it has passed all the tests on the LHS of a particular rule and its RHS is executed. The Rete algorithm remembers past test results across iterations of the rule loop. Only new facts are tested against any rule LHSs. In addition, new facts are tested against only the rule LHSs to which they are most likely to be relevant. As a result, the computational complexity is linear with respect to the size of the working memory [64].

When we increase the number of entities, the size of the working memory increases proportionally to the number of entities and their context attributes, which should be also included to the working memory. As mentioned, the increase of the working memory linearly degrades the performance of the Jess engine, which consequently increases the reaction time. In addition, event notifications are also added to the working memory. Although they are not kept for long time due to event consumption and detection window interval, the addition and removal of events requires rearranging the Rete network of nodes, which also

consumes processing time, and increases the reaction time. These behaviours are reflected on the shapes of two curves in *Figure 8-22*, namely the 45 events/s and the 90 events/s. For these curves, the reaction time is roughly linear with respect to the number of entities in the system.

In the curve 180 events/s, for more than 7500 entities, we see that the reaction time is highly unpredictable, which indicates a *saturation point*. This is explained by the fact that Jess cannot process the events fast enough to keep up with the frequency in which events are being generated (180 events/s). Therefore, event notifications may not be processed in the order they arrive, which leads to event starvation. To handle this saturation point, an event queuing event mechanism is probably necessary.

When the system gets to a saturation point, and the waiting time for processing events is bigger than the detection window interval defined for a certain rule, incorrect behaviour may possibly occur. For example, suppose we define a detection window interval of 1 second for ECARule2. Suppose that, due to system saturation, an EpilepticAlarm notification takes 3 seconds to be processed. This means that this event is discarded before it is consumed by ECARule2, because it fell out of the detection window interval. In this example, ECARule2 should have been triggered, which characterizes incorrect system behaviour. In applications that cannot afford such types of possible incorrect behaviours, we recommend defining a big detection window interval value, which can accommodate the delays caused by system saturation. If there are specific event timing constraints to be defined in a rule, such as "event1 should occur and event2 should occur at most 50 milliseconds after event1", these constraints should be included as conditions in the When clause of the rule.

### *Distributed configuration*

*Figure 8-23* depicts a graphic similar to the one in *Figure 8-22*. For this experiment we distribute the components in two different machines with 1GB memory, and 2GB memory. In the 1GB machine we run all the five context sources, and in the 2GB machine the context managers and the controller.

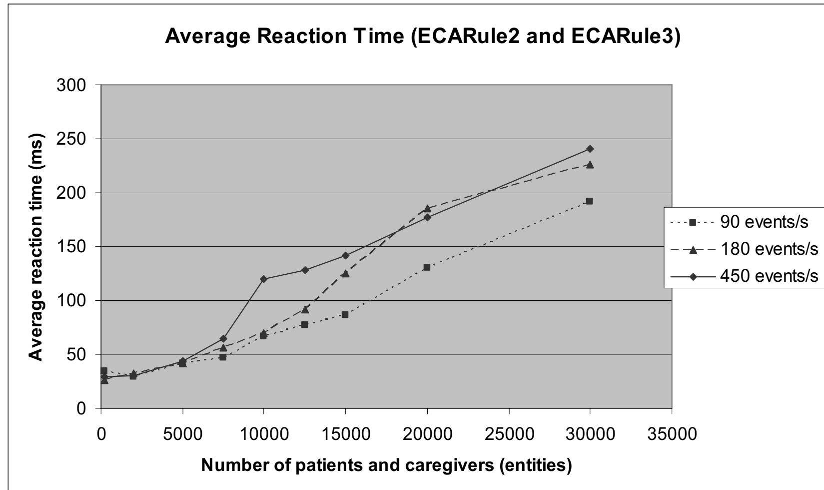



*Figure 8-23* Average reaction times for ECARule2 and ECARule3 in a distributed configuration

The graphic shows that the average reaction times in a distributed configuration improve considerably. In addition, the saturation point has not been reached even with a bigger number of entities, and a higher frequency of event generation. This behaviour can be explained by another characteristic of Jess, which is the usage of memory. Jess is said to be a "memory-intensive application" [64], since Rete is an algorithm that explicitly *trades space for speed*. Therefore, Jess performance is impacted by memory availability. In the distributed configuration, the three engines run on separate machines with increased memory availability, which improves the performance of these engines. In addition, the context sources consume a considerable amount of memory to simulate large amounts of context information and events. In the distributed configuration, the engines do not need to compete with the context sources for memory.

*Figure 8-24* compares the average reaction times for the centralized and for the distributed configurations when 90 events/s are generated. In the beginning (until 7500 entities), the centralized configuration performs better. Since at this point not much memory is required, the network *latency* for exchanging events plus the processing time required by the engines in the distributed configuration is bigger than the processing time required in the centralized configuration. Therefore, until this point, the reaction time in the centralized configuration is better than in the distributed configuration. This behaviour changes when we increase the number of entities, due to the increase in the memory usage.

Memory availability also explains why the curves in the distributed configuration are more predictable than the curves in the centralized configuration. In the centralized configuration, in which the resources are more limited, the Jess engines are more sensitive to other operating system

activities, such as threading and swapping, causing unpredictable variations in the performance.

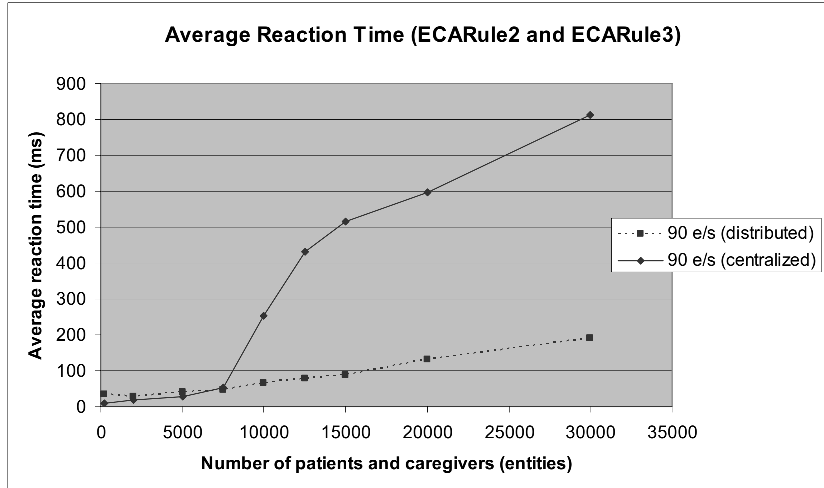



*Figure 8-24* Comparing average reaction times for the centralized and distributed configurations

## 8.4 Discussion

In this chapter we have demonstrated the feasibility of the development approach presented in this thesis. We have provided the complete design process of two different application, handling different requirements, namely the healthcare and policy management applications. In addition, we have implemented a prototype for the healthcare application, which demonstrated that the approach can be implemented in a computing system. We have also carried out evaluations that allowed us to draw relevant conclusions with respect to the performance and scalability issues. We have considered a particular medical condition in our healthcare application, namely epilepsy. Our design and prototype can be easily adapted to other medical conditions, such as heart related medical conditions. In such scenarios, a patient suffering from a heart disease can be monitored and informed of possible abnormal alterations in the heart signals.

The following paragraphs analyse our development approach on the light of the requirements defined in Chapter 2. We briefly discuss how our approach fulfils each of these requirements.

– *Support for rapid development and deployment of applications is* achieved with the systematic derivation of application realization from application specification, as discussed in 8.3.2. Since our approach alleviates application realization, it facilitates and simplifies a substantial part of the application development. In addition, application reactive

behaviours can be deployed at platform runtime, by means of ECA-DL specifications;

- *Flexibility and extensibility* is achieved with the possibility of (i) adding various application reactive behaviours at runtime; (ii) extending the platform with situation detection activities that have not been defined at specification time; and (iii) extending the platform with controller and action service components that have not been defined at design-time;
- *Support for adaptation* has been achieved by means of the situation detection framework and the controlling services. The situation detection framework allows applications not only to use context information to react on a user's request, but also to take initiative as a result of (continuously-running) context reasoning activities. In this sense, the situation framework enabled attentive application adaptation in addition to reactive application adaptation. The controlling services also perform reactivity on behalf of applications, providing explicit support for adaptation;
- *Support for distribution of context and situation reasoning* is achieved by means of distributing context sources, context managers, and controller components. In this chapter we have demonstrated distribution using the service-oriented architectural style;
- *Performance* is evaluated by collecting reaction time measurements from the healthcare prototype (section 8.3.3). Due to the Rete algorithm, Jess efficiently processes rules, which lead to satisfactory reaction times for our healthcare experiments (in the order of milliseconds). To the best of our knowledge, reaction times of this order would be acceptable for various types of context-aware applications;
- Scalability of our development approach has been assessed in the healthcare application by increasing the number of entities, the number of events generated per second, and the number of rules. We have shown that the approach scales for arbitrary number of ECA-DL rules, since ECA-DL rules can be distributed over several controller components, which can be added on demand. We have also shown that the system behaves satisfactory for a large number of entities and events.

Chapter 9

# Conclusions

In this chapter we present the conclusions of the work presented in this thesis, and we identify the topics that we recommend for future work. This chapter is further structured as follows: section 9.1 presents some general considerations; section 9.2 elaborates on the most important research contributions of this thesis; and finally, section 9.3 discusses future work.

## 9.1 General Considerations

Context-awareness has emerged as an important and desirable feature in ubiquitous applications. This feature deals with the ability of applications to utilize information about the user's environment (context) in order to tailor services to the user's current situation and needs. We have argued in this thesis that the design of context-aware applications is a challenging task, which justifies the development of novel methods, abstractions and platforms.

We aimed in this work at providing an integrated solution for the development of context-aware systems. The main objective was to facilitate the development of context-aware applications, focusing on two aspects: offering *context modelling abstractions* and providing infrastructural support by means of a *context handling platform*. Our context modelling abstractions provide application developers with proper conceptual foundations that can be extended and specialized for specific application requirements. Our context handling platform allows application functionality to be delegated to the platform, reducing application development effort, time and, therefore, costs. This allows application developers to better focus on their core business, instead of being bothered with application realization details.

We have shown that the methodology proposed in this thesis truly facilitates the development of context-aware applications. However, the success of context-awareness cannot be achieved by only using our

development approach. Results from other complementary research initiatives, which fall outside the scope of this thesis, are also necessary, such as (i) *sensor technology,* which enables various context attribute values to be gathered from the users' environments; (ii*) communication middleware*, which allows transparent distribution of system parts; (iii) *user interfaces,* which deliver context-aware services to end-users in a usable and useful way; (iv) *security and privacy,* which guarantee privacy-sensitive information to be properly used; and (v) *mobile technologies,* which allow embedding computational power in portable and mobile computers and communication devices.

## 9.2 Research Contributions

The main contributions of this thesis have been to:

- *context modelling abstractions*, which includes conceptual modelling of context and situation, and their respective realizations; and
- *a context handling platform*, which includes the generic services we have designed to support context-aware application development.

These contributions are discussed in the sequel.

### 9.2.1 Context modelling abstractions

The following aspects of context modelling have been addressed in this thesis.

***Structural context models***

The process of identifying relevant context consists of determining the "conditions" of entities in the application's universe of discourse (e.g., a user or its environment) that are relevant for a context-aware application or a family of such applications. We have argued throughout this thesis that the application's universe of discourse should be adequately characterized. As a result of this characterization process, we obtain a *conceptual model* of context. We argued that the definition of such a context model should precede the detailed design of a context-aware application.

As part of conceptual modelling of context, we have proposed basic conceptual foundations for context modelling, which allow designers of context-aware applications to represent relevant elements of a context-aware application's universe of discourse. These conceptual foundations should facilitate the specification of context models that are clearer and easier to understand. In the conceptual foundations proposed, application designers are instructed to separate the concepts of *entity* and *context.* In addition, context should be characterized as either *intrinsic* or *relational*.

Since conceptual modelling focuses on supporting structuring and inferential facilities that are psychologically grounded, the adequacy of our context modelling technique is determined by its contribution to common understanding of context among the stakeholders of a context-aware application (e.g., users and designers). Therefore, we have justified our modelling choices with results from foundational ontologies, which are in line with conceptual theories in philosophy and cognitive sciences.

***Situation models***

Our structural context models allow application designers to represent all possible state-of-affairs of an application's universe of discourse, without discriminating particular situations that may be of interest to applications. In order to explicitly discriminate particular state-of-affairs of interest, we have introduced the concept of *situation*. A situation is a composite concept whose constituents are (a combination of) entities and their context conditions. Situations extend context models since they can be composed of more elementary kinds of context conditions, and in addition can be composed of existing situations themselves.

We have proposed a novel model-based approach for the specification of situations. Situations are specified using standard UML 2.0 class diagrams enriched with OCL 2.0 constraints to define the conditions under which situations of a certain type are allowed to exist. We support a wide range of situations, which can be composed of more elementary kinds of context and situations. This allows modularization of the situation models, improving organization and reuse of situation specifications. We have also introduced situation *chronoids*, which allow us to explicitly capture past and present situations.

***Context and situation information models***

We have distinguished the concepts of *context* and *context information* in our approach. We regard context as the real world phenomena, while context information refers to the representation of (constituents of) context in a software application, such that this representation can be manipulated and exchanged. We have provided support for bridging the gap between conceptual *context models* and *context information models*, by means of serializable measurement datatypes.

As part of the context information modelling phase we consider (i) how context is sensed; (ii) how context information is produced, learned, inferred and used, and (iii) the validity and *Quality of Context (QoC)* information. QoC is concerned with meta-information that describes the quality of the context information. We have discussed three QoC parameters, namely, *precision*, *probability of correctness* and *freshness*.

***Context and situation realization***

We have proposed a novel model-driven approach for the realization of situations in context-aware applications. This realization is rule-based, and executes on mature and efficient rule engine technology available off-the-shelf. The rule set for situation detection is derived systematically from the UML and OCL situation specifications and is deployed directly in the rule-based engine. In order to perform situation detection rules, we use a general-purpose rule-based platform, namely Jess.

The use of a rule-based approach in the situation detection framework allows applications not only to use context information to react on a user's request, but also to take initiative as a result of (continuously-running) context reasoning activities. In this sense, our situation detection mechanism enables *attentive* application adaptation in addition to *reactive* application adaptation. The rule-based approach also facilitates the generation of special events that are created when situations begin and cease to hold. These events are called *enter true* and *enter false* events, respectively.

We have argued that a distributed solution to situation detection has benefits, which apply particularly to context-aware applications. We have realized communication between rule engines in two different ways, namely by using a generic rule-based distribution middleware, and by following a service-oriented architecture style.

### 9.2.2 The context handling platform

We have argued throughout this thesis that it is not cost-effective to build each individual context-aware application from scratch. It is also too complex for each individual application to capture and process context information just for its own use. In addition, we have observed that similar functions are inefficiently implemented in different context-aware applications or family of applications. Therefore, we have concluded that a number of commonly used functions can be made available for *reuse* to various context-aware applications, by means of *generic services,* which are offered by *generic components.*

In order to cope with complexity and cost-effectiveness of building context-aware applications, we have proposed a context handling platform that provides context-aware generic services, i.e. services that support context-aware applications, regardless of the application domain. As part of the platform, we have defined *discovery*, *context provisioning*, *controlling* and *action services*, which can be combined and configured to satisfy application specific requirements. Particularly, in this thesis we have focused on the context provisioning and the controlling services.

***Context-aware architectural patterns***

In this thesis we have proposed three architectural patterns that can be beneficially applied in the development of context-aware services platform, namely the *Event-Control-Action* pattern, the *Context Sources and Managers Hierarchy* pattern and the *Actions* pattern. By decoupling context concerns from action concerns, the Event-Control-Action pattern has effectively enabled the distribution of responsibilities among various business parties in a context-aware services platform. Applying such design principles greatly improves the extensibility and flexibility of the platform, since context processors and action components can be developed and deployed separately and on demand. In addition, the definition of application behaviour by means of condition rules allows the dynamic deployment of context-aware application behaviours and permits the configuration of the platform at runtime.

The Context Sources and Managers Hierarchy pattern enables flexible and dynamic distribution of context information processing activities within a collaborative network of context sources and managers. This approach has enabled encapsulation and a more effective, flexible and decoupled distribution of context processing activities (sensing, aggregating, inferring and predicting). This approach improves collaboration among context information owners and it is an appealing invitation for new parties to join this collaborative network, since collaboration among more partners enables the availability of potentially richer context information.

Finally, the Actions pattern defines a structure of action performer components, which enables the coordination of compound actions and the separation of abstract action purposes from their implementations. This allows the dynamic selection of action implementations at runtime, improving the extensibility and flexibility of the platform. It also allows 3rd party services to be developed and deployed on demand at platform runtime.

***Context handling platform architecture***

We have proposed an architecture for the context handling platform that applies our context-aware architectural patterns. In this architecture, we define context processor, controller, and action components.

Context processor components gather context information from the user's environment. Based on context information measurements, context processor components perform context reasoning and generate context and situation events, which are offered to the other components of the platform and to application-specific components.

The controller component aims at executing particular context-aware application-specific behaviours within the platform. In this scope, context-aware application behaviours can be described as logic rules, which are

called Event-Condition-Action (ECA) rules that are consistent with the Event-Control-Action pattern. In order to realize application behaviours, the controller component gathers context from context processor components. When the combination of context conditions defined by the application-specific behaviours is met, the controller component invokes the required actions on the action components.

***Controlling services***

The controlling services are the services offered by controller components, which aim at executing particular application behaviours in the platform. The controller component implements a rule engine that can efficiently process rules, which are matched against various types of events, context, and situation conditions. When events have occurred and conditions hold for a rule, the action part of the rule is executed, which consists of various types of service invocations. In order to facilitate the specification of context-aware reactive behaviours, we have developed *ECA-DL*, a domain specific language for specifying context-aware reactivity. In order to demonstrate the suitability of ECA-DL rules, we have used the Jess engine as the ECA-DL execution environment. Since only Jess rules are accepted by the Jess engine, we have provided a mapping framework that can be used to generate Jess rules from ECA-DL rules.

We have discussed two approaches for the automation of the mappings from ECA-DL rules to Jess rules, namely the *parser* and the *MDA approach*. The parser approach implements a parser that breaks down an ECA-DL rule into indivisible elements, which are defined in the ECA-DL metamodel. For each of these elements, Jess expressions are generated. The MDA approach generates Jess rules from ECA-DL based on transformations defined in terms of elements of the metamodels of those languages. The aim of this solution is to *formalize* the mapping process from ECA-DL rules to Jess rules and implement this mapping by applying MDA technologies.

The controller component has effectively improved the flexibility, extensibility and adaptability of the platform. Higher flexibility and extensibility have been achieved by allowing arbitrary application logics to be deployed at platform runtime. This enables the platform to match applications' requirements at runtime, which have not been anticipated at platform design-time. Finally, higher platform adaptability has been achieved since the controller is capable of reacting on both events and conditions by executing actions that allow adaptation of components according to context.

***Structural design of context-aware applications***
We have proposed a design methodology for structuring context-aware applications, which is based on the service-oriented architectural style. Service-oriented architecture is a design discipline in which applications are organized as compositions of services. At the beginning of the design process, we focus on the application service as whole, i.e. on the external perspective of the application, in terms of the behaviour that can be experienced by the environment (end-users) of the application. Following this design process, we gradually refine the application service into sub-services, until the system can be finally implemented in a computing system.

Since applications in our approach are developed with the support of the platform, application developers should be able to identify the application-specific functions, and the generic functions that can be performed by the platform. The application-specific services are implemented by the application developers themselves, and the generic functions are provided by the platform components.

We have demonstrated the suitability of our development approach by means of case studies. We have designed the healthcare and the policy management application following the design process proposed in this thesis, and we have implemented a prototype for the healthcare application. The application design process included the activities (i) context modelling, (ii) context information modelling; (iii) application structural design; (iv) context provisioning services design; and (v) application deployment on the controlling service. In addition, we have successfully assessed the performance and scalability of the prototype. Performance has been evaluated using the reaction time parameter, which indicates the period of time between generating an event and invoking an action. Our experiments showed that the healthcare application performed satisfactory for a large number of users and events.

## 9.3 Future Work

We have identified future research in the following areas:

- *handling context and situation*, which is related to our context and situation modelling approach;
- *automation processes*, which is related to the automation of specifications to realizations;
- *context handling platforms*, which is related to the components and services offered by the platform.

### 9.3.1 Handling context and situations

Since sensor technology is imperfect by definition, as part of our context and situation modelling abstractions, we have defined simple quality parameters, which specify the quality of the context information with respect to the probability of correctness, freshness and precision aspects. We have not discussed in this thesis how Quality of Context (QoC) is evaluated, how quality of situation information can be derived from QoC information, and how to improve QoC using, for example, redundancy of context sources. These aspects of QoC should be handled by realistic context-aware application, which requires further investigation. In addition, we have not incorporated QoC in our healthcare prototype. Extensions to our approach should also include the implementation of QoC abstractions.

In this thesis we have studied context reasoning by means of the situation detection framework, which detects particular (past or current) situations of interest. We have not tackled reasoning activities using learning and prediction techniques. Past occurrences of context and situation information can be extrapolated to predict future user's and application's behaviours. Similarly, pattern of situation occurrences could be used in a learning process in order to anticipate all kinds of application behaviours. Future research is necessary to incorporate learning and prediction in our context modelling abstractions.

In future work, one should study more complex mechanisms for discarding historical situation records that will no longer be used. Our current solution uses time-to-live for discarding historical records. An alternative solution is to eliminate all historical data that is not referred to by any active situation. This requires inspection on situation type dependencies, which tends to be complex.

### 9.3.2 Automation processes

In the realization approach for both situation detection and ECA-DL rule execution, we have considered Jess as the underlying technology. In order to map situation and ECA-DL rule specifications to Jess code, we provide mappings that can be used to systematically derive rules from UML, OCL and ECA-DL specifications. We have partially automated the mappings from ECA-DL specifications to Jess. As part of future work, our approach should be extended to provide automated transformations from specifications to realizations for both the situation detection and ECA-DL rules. Preferably, the automation processes should be based on the MDA approach, in order to facilitate (i) the maintenance of rules whenever a newer version of the Jess engine is released; and (ii) the implementation of ECA-DL rules in a different rule-based platform.

Finally, future measurement datatypes, context source, and context manager components should be automatically generated from the conceptual models. As we have argued, our approach provides guidelines to fill the gap between conceptual models and context information models (measurement dataypes). We have noticed that the derivation of the information models from the conceptual models follows certain patterns. Therefore, it should be possible to automatically derive the measurement datatypes from the conceptual models.

Analogously, we have noticed that context sources and managers can be systematically derived from the measurement datatypes these components are supposed to provide. Therefore, it should be possible to automatically generate context sources from context measurement datatypes, and context managers from situation measurement datatypes.

### 9.3.3 Context handling platform

We have focused on two specific types of services offered by the platform, namely context provisioning and controlling services. For scoping reasons, two types of services have been explicitly disregarded in this thesis, namely discovery services, and action services. A discovery service allows distributed services that are not known beforehand to be found using various characteristics of the service, such as type of the information offered by the service, quality of the information, quality of the service, and so forth. Parallel efforts have been carried out in the field of context-aware service discovery [54, 99]. Our current simplified discovery mechanism could be extended with these efforts in order to enable richer discovery of context provisioning, controlling, actions and application-specific services, at platform runtime.

The action services we have presented in this thesis offer simple actions, which can be invoked from the controller component independently from each other. In practice, actions often have interdependencies, such as when an action can only be enabled when another action has been successfully executed. The action services we propose could be extended to allow complex compositions and interdependencies of action services. Future work could investigate how ECA-DL can be extended to allow action composition specification based on standard service composition specification languages, such as BPEL [11].

With respect to quantitative crosscutting issues, we have assessed performance and scalability, but we have not considered the *reliability* of the context handling platform components. In future work, one could study how to tackle reliability in case unexpected circumstances occur, such as when context provisioning or controlling services stop working. A simple

solution to improve reliability may consist of using redundancy of components. Further investigation in this area is necessary.

We have briefly discussed in this thesis the importance of privacy, security and trust issues. Given the privacy sensitive nature of context information, we believe that the acceptance of context-awareness by application users will heavily rely on how users can trust that the applications use their context information in a private and secure way. These issues are also identified for future work.

Finally, in order to demonstrate the suitability of the platform in other application domains, context-aware applications should be developed on top of our platform for different application areas, such as leisure, government, and banking.

Appendix A

# Mappings from OCL to Jess

This appendix presents how particular OCL constructs can be systematically mapped to Jess constructs.

***Navigation***

In UML, it is possible to navigate from object to object by means of their associations, using *dots*. For example, if $object_1$ is associated with $object_2$, we can navigate to $object_2$ through $object_1$, such as $object_1.object_2$. In the implementation this is also possible, since $object_2$ is implemented as an attribute of $object_1$. An example of navigation from our context model is person.hasGeoLocation.GeoCoordinates.latitude, which allows us to access the location's latitude value of a person.

For each object and its respective attribute (navigation), one line in the Jess language is necessary. For example, we would need three lines in the Jess language to navigate to a person's latitude value. The first line to navigate to the person's location object (person.hasGeoLocation), the second to navigate to the geolocation coordinates (person.hasGeoLocation.GeoCoordinates) and finally the third line to navigate to the latitude value (person.hasGeoLocation.GeoCoordinates.latitude).

Consider a simple example in which we access an attribute of an object ($object_1$), which is also an object ($object_2$), as opposed to a primitive type (e.g., numbers and strings). In OCL this navigation is represented in the following way: $object_1.object_2$. In Jess, we would represent this example as follows: ($ObjectType_1$ (OBJECT ?$object_1$) ($object_2$ ?$object_2$)), where (i) ObjectType refers to the type of object object, (ii) OBJECT is the special slot which refers to a Java object, and finally (iii) the slot $object_2$ representing the name of the attribute, in which ?$object_2$ is a variable containing a reference to the object itself.

Consider yet another example of a primitive datatype attribute (e.g., a numeric value) object.pdatatype, where pdatatype refers to the attribute name. In Jess we would navigate to this attribute by defining (ObjectType (OBJECT ?object)

(pdatatype ?pdatatype)), in which ?pdatatype is a variable containing the value of the attribute. *Table A-1* depicts some examples of navigation and their corresponding statements in the Jess language.

*Table A-1* Examples of navigation mappings

| OCL language | Jess language |
|---|---|
| object | (ObjectType (OBJECT ?object)) |
| object.pdatatype | (ObjectType (OBJECT ?object) (pdatatype ?pdatatype)) |
| $object_1.object_2$ | ($ObjectType_1$ (OBJECT ?$object_1$) ($object_2$ ?$object_2$)) |
| $object_1.object_2.object_3$ | ($ObjectType_1$ (OBJECT ?$object_1$) ($object_2$ ?$object_2$))<br>($ObjectType_2$ (OBJECT ?$object_2$) ($object_3$ ?$object_3$)) |
| $object_1$.datatype | ($ObjectType_1$ (OBJECT ?$object_1$) (datatype ?pdatatype)) |
| $object_1$.datatype.pdatatype | ($ObjectType_1$ (OBJECT ?$object_1$) (datatype ?datatype))<br>(DataType (OBJECT ? datatype) (pdatatype ?pdatatype)) |
| $object_1.object_2$. datatype. pdatatype | ($ObjectType_1$ (OBJECT ?$object_1$) ($object_2$ ?$object_2$))<br>($ObjectType_2$ (OBJECT ?$object_2$) (datatype ?datatype))<br>(DataType (OBJECT ?datatype) (pdatatype ?pdatatype)) |
| object->collection | (ObjectType (OBJECT ?object) (collection ?collection) |

When navigating from one object to another (e.g., $object_1.object_2$), only one line in Jess is necessary, since the second object is an attribute (slot) of the first. When navigating through three objects, in which $object_3$ is an attribute of $object_2$, which in turn is an attribute of $object_1$ ($object_1.object_2.object_3$), two lines are necessary in Jess, one line to access $object_1$ and its attribute $object_2$, and the other line to access $object_2$ and its attribute $object_3$.

An example of navigation through a datatype (e.g., GeoLocationCoordinates) to a primitive datatype (e.g., latitude, which is a numeric value) is represented in the following way $object_1$.datatype.pdatatype (see *Table A-1*). Since a datatype is implemented as normal class in the Java implementation, the mapping to Jess is the same for $object_1.object_2$.pdatatype, i.e. the expression ($ObjectType_1$ (OBJECT ?$object_1$) (datatype ?datatype)) is used to navigate to the datatype object, and (DataType (OBJECT ? datatype) (pdatatype ?pdatatype)) is used to navigate to the value of the pdatatype attribute.

The last example of *Table A-1* shows the mapping for the navigation to a collection attribute (object->collection). Similar to the other examples, in Jess this would be mapped to (ObjectType (OBJECT ?object) (collection ?collection)). The mechanism we have used to manipulate collections is discussed in the following sections.

### *Equality*

In general, equality in the OCL language maps to equality between slot values. The simplest example of equality between two objects, $object_1$ and $object_2$ ($object_1 = object_2$), maps to the following in Jess: the first line navigates

to $object_1$ (($ObjectType_1$ (OBJECT $object_1$))), and the second line navigates to $object_2$ and tests whether this object is equal to $object_1$, like in ($ObjectType_2$ (OBJECT $object_2$&:(eq (?$object_2$ ?$object_1$)))). We use the Jess function eq (expr expr*) to test equality between objects and strings, which returns TRUE if the first argument is equal in type and value to all subsequent arguments. Similar functions (eq* or =) may be used to check equality for numeric values. The combination of symbols &: reads "such that" in Jess. So, the code (…?$object_2$&:(eq (?$object_2$ ?$object_1$))) reads "object 2 such that object 2 is equal to object 1". In fact, the ampersands symbol (&) represent the logical "and", and the colon symbol (:) followed by a functional call performs a test in the LHS of a Jess rule, in which the test succeeds if the function returns TRUE. These symbols together (&:) allow one to name a variable and to perform a functional test on this variable. *Table A-2* depicts examples of equality mappings.

*Table A-2* Examples of equality mappings

| OCL language | Jess language |
|---|---|
| $object_1$ = $object_2$ | ($ObjectType_1$ (OBJECT ? $object_1$))<br>($ObjectType_2$ (OBJECT ?$object_2$&:(eq (?$object_2$ ?$object_1$)))) |
| $object_1$. $object_2$= $object_3$ | ($ObjectType_1$ (OBJECT ?$object_1$) ($object_2$? $Object_2$))<br>($ObjectType_3$ (OBJECT ?$object_3$&:(eq (?$object_3$ ?$object_2$))) |
| $object_1$. $object_2$= $object_3$.$object_4$ | ($ObjectType_1$ (OBJECT ?$object_1$) ($object_2$ ?$object_2$))<br>($ObjectType_3$ (OBJECT ?$object_3$) ($object_4$ ?$object_4$&:(eq (?$object_4$ ?$object_2$)))) |
| $object_1$.pdatatype = number | ($ObjectType_1$ (OBJECT ?$object_1$) (pdatatype ?pdatatype&: (= (?pdatatype number)))) |
| $object_1$.pdatatype = $object_1$.datatype.pdatatype | ($ObjectType_1$ (OBJECT ?$object_1$) (pdatatype ? $pdatatype_1$))<br>($ObjectType_1$ (OBJECT ?$object_1$) (datatype ?datatype))<br>(DataType (OBJECT ?datatype) (pdatatype ? $pdatatype_2$ &: (eq (?$pdatatype_2$ ?$pdatatype_1$)))) |
| $object_1$->$collection_1$ = $object_2$-> $collection_2$ | ($ObjectType_1$ (OBJECT ?$object_1$) (collection ?$collection_1$)<br>($ObjectType_2$ (OBJECT ?$object_2$) (collection ?$collection_2$&: (eq (?$collection_1$ ?$collection_2$)))) |

The mapping of other comparison infix operators (>, >=, <, <=, <>) is similar to the equality operator, except that they only work on numeric values. For example, the expression $object_1$.pdatatype > number is mapped to ($ObjectType_1$ (OBJECT ?$object_1$) (pdatatype ?pdatatype&: (> (?pdatatype number)))).

### *Pattern matching logical operators*

In Jess, there are two sets of logical operators, namely the pattern matching logical operators (called conditional elements), and the Boolean logical operators. The conditional elements (e.g., and, or and not) operate on patterns. For example, the expression not (pattern) returns true if there is no

match for pattern in the working memory. The Boolean logical operators (e.g., &, ! and |), in contrast, operate on Boolean expression, as opposed to patterns. For example, the expression !x returns true if the value of x is false. The following conditional elements are supported in our implementation:

- and: matches multiple facts in the working memory. The and conditional element can combine any number of patterns, and the resulting pattern is matched if and only if all enclosed patterns are matched. For example, the resulting pattern of and (pattern1 pattern2) matches if both patterns are matched in the working memory;
- or: matches alternative facts. The or conditional element can combine any number of patterns, the resulting pattern is matched if at least one of these patterns is matched. For example, the result pattern of or (pattern1 patter2) is matched when at least one patterns is matched in the working memory;
- not: matches if no facts match. The not conditional element is a unary operator, i.e. it matches one pattern only. For example, the result of not (pattern) is matched when the pattern pattern is not matched in the working memory.

As we have already mentioned, the outer most occurrences of the AND logical operators in OCL are mapped to lines in the Jess rules. However, inner occurrences of AND (<expr$_1$> AND <expr$_2$>), for example, as an operand of the OR logical operator, are mapped to an and ((expr$_1$) (expr$_2$)) in Jess. The <expr$_1$> OR <expr$_2$> logical operator, in contrast, is always mapped to an or ((expr$_1$) (expr$_2$)) in Jess. Finally, the NOT <expr> logical operator is mapped to not (expr) in Jess. *Table A-3* depicts examples of mappings from OCL expressions using logical operators, to conditional elements in Jess.

*Table A-3* Examples of mappings with logical operators

| OCL language | **Jess language** |
|---|---|
| object$_1$ AND object$_2$ | (ObjectType$_1$ (OBJECT ?object$_1$))<br>(ObjectType$_2$ (OBJECT ?object$_2$)) |
| (object$_1$.pdatatype = number)<br>AND<br>(object$_2$.pdatatype = number) | (and (ObjectType$_1$ (OBJECT ?object$_1$)<br>(pdatatype ?pdatatype&:(= ?pdatatype number)))<br>(ObjectType$_2$ (OBJECT ?object$_2$)<br>(pdatatype ?pdatatype&:(= ?pdatatype number)))<br>) |
| object$_1$ OR object$_2$ | (or (ObjectType$_1$ (OBJECT ?object$_1$))<br>(ObjectType$_2$ (OBJECT ?object$_2$))<br>) |
| (object$_1$. pdatatype = value)<br>OR<br>(object$_2$. pdatatype = value) | (or (ObjectType$_1$ (OBJECT ? object$_1$)<br>(pdatatype ?pdatatype&:(eq (?pdatatype value))))<br>(ObjectType$_2$ (OBJECT ? object$_2$)<br>(pdatatype ?pdatatype&:(eq (?pdatatype value))))<br>) |

| ((object$_1$.object$_3$= object$_4$) AND<br>(object$_4$. pdatatype = value))<br>OR<br>((object$_5$.object$_6$= object$_7$) AND<br>(object$_7$. pdatatype = value)) | (or (and (ObjectType$_1$ (OBJECT ? object$_1$) (object$_3$ ? object$_3$))<br>(ObjectType$_4$ (OBJECT ? object$_4$&:<br>(eq (?object$_4$ ? object$_3$)))<br>(ObjectType$_4$ (OBJECT ?object$_4$)<br>(pdatatype ?pdatatype&:(eq(?pdatatype value))))<br>)<br>( and (ObjectType$_5$ (OBJECT ? object $_5$) (object$_6$ ?object $_6$))<br>(ObjectType$_7$ (OBJECT ? object$_7$&:<br>(eq (?object7 ?object$_6$))<br>(ObjectType$_7$ (OBJECT ?object$_4$)<br>(pdatatype ?pdatatype&:<br>(eq (?pdatatype value))))<br>)<br>) |
|---|---|
| NOT ((object$_1$.object$_3$= object$_4$) AND<br>(object$_4$. pdatatype = value)) | not (and (ObjectType$_1$ (OBJECT ? object$_1$) (object$_3$ ? object$_3$))<br>(ObjectType$_4$ (OBJECT ? object$_4$&:<br>(eq (?object$_4$ ? object$_3$))<br>(ObjectType$_4$ (OBJECT ?object$_4$)<br>(pdatatype ?pdatatype&:(eq(?pdatatype value)))<br>) |

***Method invocations***

Method invocations in OCL are mapped to method invocations on Java objects using the Jess function call. If the method requires parameterization, the parameters are resolved beforehand. The first argument of call is a Java object, and the second argument is the name of the method to invoke. The next arguments to call are the arguments to be passed as parameters to the method. The Test conditional element is used to check the value of a Boolean expression, which could be the result of a method invocation, or the result of a comparison. For example, the OCL invariant object$_1$->method (object$_2$), maps to the following lines to in Jess:

- (ObjectType$_1$ (OBJECT ?object$_1$)): navigates to object 1;
- (ObjectType$_2$ (OBJECT ?object$_2$)): navigates to object 2, which is an argument of the method;
- (test (call ?object1 method ?object2)): invokes the method on object1 using the call function, passing object2 as argument. The test conditional element checks whether the return of this method is true.

It is also possible to compare the result of a method invocation using comparison functions, such as >, and <. For example, the mapping of expression object$_1$->method (object$_2$) > number is almost identical the previous example, except that test now checks whether the result of (> (call ?object$_1$ method ? object$_2$) number) is true. *Table A-4* depicts examples of mappings from OCL expressions using method invocations, to expressions in Jess.

*Table A-4* Examples of mappings with method invocations

| OCL language | Jess language |
|---|---|
| object$_1$->method (object$_2$) | (ObjectType$_1$ (OBJECT ?object$_1$))<br>(ObjectType$_2$ (OBJECT ?object$_2$))<br>(test (call ?object$_1$ method ?object$_2$)) |
| object$_1$.object$_2$->method (object$_1$.pdatatype) | (ObjectType$_1$ (OBJECT ?object$_1$) (object$_2$ ?object$_2$)<br>(ObjectType$_1$ (OBJECT ?object$_1$) (pdatatype ?pdatatype))<br>(test (call ?object$_2$ method ?pdatatype)) |
| object$_1$.object$_2$->method (object$_1$.pdatatype, value) | (ObjectType$_1$ (OBJECT ?object$_1$) (object$_2$ ?object$_2$)<br>(ObjectType1 (OBJECT ?object$_1$) (pdatatype ?pdatatype))<br>(test (call ?object$_2$ method ?pdatatype ?value)) |
| object$_1$->method (object$_1$.pdatatype, object$_2$.object$_3$) | (ObjectType$_1$ (OBJECT ?object$_1$))<br>(ObjectType$_1$ (OBJECT ?object$_1$) (pdatatype ?pdatatype))<br>(ObjectType$_2$ (OBJECT ?object$_2$) (object$_3$ ?object$_3$))<br>(test (call ?object$_1$ method ?pdatatype ?object$_3$)) |
| object$_1$->method (object$_2$) > number | (ObjectType$_1$ (OBJECT ?object$_1$))<br>(ObjectType$_2$ (OBJECT ?object$_2$))<br>(test > (call ?object$_1$ method ?object$_2$) number) |
| object$_1$->method (object$_2$) = object$_3$ | (ObjectType$_1$ (OBJECT ?object$_1$))<br>(ObjectType$_2$ (OBJECT ?object$_2$))<br>(ObjectType$_3$ (OBJECT ?object$_3$))<br>(test (eq (call ?object$_1$ method ?object$_2$) ?object$_3$)) |
| ObjectType::StaticMethod (object$_2$) | (ObjectType$_2$ (OBJECT ?object$_2$))<br>(test (call ObjectType StaticMethod ?object$_2$)) |

***Boolean and arithmetic operators***

The Boolean logical operators (AND, OR, NOT) in OCL used in method arguments are mapped to &, |, ~ in Jess, respectively. In OCL, it is possible to pass complete Boolean expression as method arguments; for example, object->method (object.pdatatype = value$_1$) OR (object.pdatatype = value$_2$) is a valid OCL expression. In Jess, the OR logical operator would be mapped to a Boolean logical operator, as opposed to a conditional element, since the application of the OR in this case should return either false or true, and not a pattern match. The mapping in Jess of an or, in which the first two lines aim at navigating to object$_1$ and its pdatatype attribute is the following.

```
(ObjectType1 (OBJECT ?object1))
(ObjectType1 (OBJECT ?object1) (pdatatype ?pdatatype))
(test (call ?object1 method (eq (?pdatatype value1) | (eq (?pdatatype value2))))
```

The third line invokes the method, and tests whether the return of the method is true. The argument passed to the method is a Boolean value, which is true if the value of the pdatatype is either equal to value$_1$ or value$_2$.

*Table A-5* depicts examples of mappings from OCL expressions using Boolean expressions in method parameterizations, to expressions in Jess.

*Table A-5* Examples of mappings with Boolean operators

| OCL language | Jess language |
|---|---|
| $object_1$->method ($object_1$.datatype.pdatatype and ($object_2$.datatype.pdatatype = value)) | ($ObjectType_1$ (OBJECT ?$object_1$))<br>($ObjectType_1$ (OBJECT ?$object_1$) ($datatype_1$ ?$datatype_1$))<br>($DataType_1$ (OBJECT ?$datatype_1$) ($pdatatype_1$ ?$pdatatype_1$))<br>($ObjectType_2$ (OBJECT ?$object_2$))<br>($ObjectType_2$ (OBJECT ?$object_2$) (datatype2 ?$datatype_2$))<br>($DataType_2$ (OBJECT ?$datatype_2$) ($pdatatype_2$ ?$pdatatype_2$))<br>(test (call ?$object_1$ method (eq (?pdatatype1 $value_1$) & (eq (?$pdatatype_2$ $value_2$)))) |
| $object_1$->method (not $object_2$.datatype.pdatatype) | ($ObjectType_1$ (OBJECT ?$object_1$))<br>($ObjectType_2$ (OBJECT ?$object_2$) (datatype ?datatype))<br>(DataType (OBJECT ?datatype) (pdatatype ?pdataype))<br>(test (call ?$object_1$ method (~?pdataype)) |

The OCL arithmetic operators +, -, * and / are mapped to their corresponding operators in the Jess language (+, -, * and /), using the usual form of a function call. For example, the OCL expression number + number is mapped to (+ number number) in Jess. The same applies to the other arithmetic operators.

### *Temporal attributes*

In our approach, temporal aspects are implemented by using situation initial and final time attributes, which are defined as the Java datatype Date in the implementation. Therefore, the operations performed on these attributes in the OCL are mapped to the operations supported by the Date datatype. *Table A-6* depicts some examples of mappings with temporal attributes. The method getTime returns the number o milliseconds represented by a Date object, since January first, 1970. The methods after and before test whether some date is after or before some other date, respectively.

*Table A-6* Examples of mappings with temporal attributes

| OCL language | Jess language |
|---|---|
| $object_1$.initialtime->after ($object_2$.finaltime) | ($ObjectType_1$ (OBJECT ?$object_1$) (initialtime ?initialtime))<br>($ObjectType_2$ (OBJECT ?$object_2$) (finaltime ?finaltime))<br>(test (call ?initialtime after ?finaltime)) |
| $Objec_1$.initialtime->before ($object_2$.finaltime) | ($ObjectType_1$ (OBJECT ?$object_1$) (initialtime ?initialtime))<br>($ObjectType_2$ (OBJECT ?$object_2$) (finaltime ?finaltime))<br>(test (call ?initialtime before ?finaltime)) |

| | |
|---|---|
| $object_1$.initialtime - $object_2$.finaltime < 60000 | ($ObjectType_1$ (OBJECT ?$object_1$) (initialtime ?initialtime)) ($ObjectType_2$ (OBJECT ?$object_2$) (finaltime ?finaltime)) (test (< (- (call ?initialtime getTime) (call ?finaltime getTime)) 60000))) |
| $object_1$.initialtime + $object_2$.finaltime = 3000 | ($ObjectType_1$ (OBJECT ?$object_1$) (initialtime ?initialtime)) ($ObjectType_2$ (OBJECT ?$object_2$) (finaltime ?finaltime)) (test (= (+ (call ?initialtime getTime) (call ?finaltime getTime)) 30000))) |
| Time->now() | (call System currentTimeMillis) |

***Collections***

Collections in our approach are only manipulated by means of special manipulation methods, which are implemented in the classes holding the collection attributes. Our mappings do not support native manipulation of collections offered by OCL. Consider the following example, which specifies that the situation (SituationInChat) exists when a person named John is involved in three or more chats. This can be useful, for example, to refuse further incoming invitations for chatting.

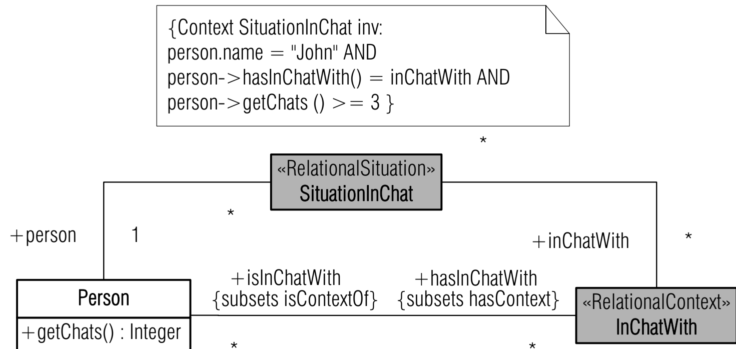



*Figure A-25* SituationInChat specification

The subsets association hasInChatWith is implemented in class Person by the method hasInChatWith, which returns a collection of inChatWith objects, each one representing a chat in which John is participating. The method getChats returns the number of chats John is participating. Implementing these methods is the responsibility of the application developer. These methods may be specified already at design time, together with the situation models.

Since manipulating collections is realized by means of method invocations, the mapping to the jess language follows the mapping of method invocations discussed in the previous sections (*Table A-4*). For example, the invariant of *Figure A-25* is mapped to the EnterTrue rule depicted in *Figure A-26*.

*Figure A-26* SituationInChatWith realization in Jess

```
;EnterTrue (SituationInChatWith)
(defrule entertrue_situation_inchatwith                Manipulating the collection
(defrule entertrue_situationinchat
   (Person (OBJECT ?person) (identity "John"))

   (Person (OBJECT ?person) (hasInChatWith ?hasInChatWith))
   (test (>= (call ?person getChats) 3))

   (not (SituationInChat (OBJECT ?SituationInChat)(person ?person)(finaltime nil)))
   =>
   (bind ?SituationInChat (new situation_control.SituationInChat ?person))
   (definstance SituationInChat ?SituationInChat)
)
```

The equality person->isInChatWith() = inChatWith is mapped to (Person (OBJECT ?person) (isInChatWith ?inChatWith)), i.e. inChatWith is omitted in the Jess rule, and the collection person->isInChatWith is mapped as a normal collection, as depicted in *Table A-1*.

Since collections are manipulated by means of methods implemented in the classes holding the collection attribute, the inChatWith collection would normally be manipulated by methods implemented in the SituationInChat class, which is the reference class of this invariant. However, at runtime, instances of SituationInClass initially do not exist, since they are created when the OCL invariant starts to hold. Therefore, it is not possible to invoke methods of SituationInClass from the EnterTrue Jess rule, simply because initially there is no instance of SituationInClass in the working memory. For this reason, the inChatWith collection is not mapped to Jess code. This imposes no limitations to the mapping because the associations between situation type classes and context type classes are also represented between entities type classes and their respective context types.

The method invocation person->getChats (inChatWith), on the contrary, is mapped as a normal method invocation, as depicted in *Table A-4*. *Table A-7* depicts examples of how collections are mapped to the Jess language.

*Table A-7* Examples of mappings with collections

| OCL language | Jess language |
|---|---|
| object->collection | (ObjectType (OBJECT ?object) (collection ?collection)) |
| object->$collection_1$ = $collection_2$ | (ObjectType (OBJECT ?object) (collection $?collection_1$)) |
| $object_1$->$collection_1$ = $object_2$-> $collection_2$ | ($ObjectType_1$ (OBJECT $?object_1$) (collection ?collection1) ($ObjectType_2$ (OBJECT $?object_2$) (collection $?collection_2$&: (eq ($?collection_1$ $?collection_2$)))) |

### *OclIsUndefined*

The OCL operation oclIsUndefined() is part of the OCL standard library and tests whether the value of an expression is undefined. This is mapped to the Jess not conditional element, which pattern matches if a fact does not exist

in the working memory. In case the oclIsUndefined() is used in combination with not Boolean operator, such as not oclIsUndefined(), this maps to a normal pattern matching in the rules, since it checks whether an expression is existent in the working memory.

For example, the expression object->oclIsUndefined(), maps to (not (ObjectType (OBJECT ?object)) in Jess language, and the expression not object.oclIsUndefined() maps to (ObjectType (OBJECT ?object)).

### *Example*

*Figure A-27* depicts the EnterTrue Jess rule for SituationContained, depicted in *Figure 6-1*. The OCL invariant for this situation specification is shown in the sequel. The numbers depict the correspondences between the OCL specification and the Jess code.

```
Context SituationContained inv:
❶ person.hasGeoLocation = locationPerson AND
❷ building.hasGeoLocation = locationBuilding AND
❸ building.hasSpatialCoordinates = spatialCoord AND
❹ (spatialCoord.dimension->containment (locationPerson.coordinates,
        locationBuilding.coordinates, spatialCoord.dimension)) AND
❺ building.ID = "Zilverling"
```

*Figure A-27* SituationContained realization in Jess

```
(defrule entertrue_situation_contained
❶ (Person (OBJECT ?person)( hasGeoLocation ? person_hasGeoLocation))
  (GeoLocation (OBJECT ?locationPerson&:(eq ?locationPerson ? person_hasGeoLocation)))

❷ (Building (OBJECT ?building)(geoLocation ?building_hasGeoLocation))
  (GeoLocation (OBJECT ?locationBuilding&:(eq ?locationBuilding ?building_hasGeoLocation)))

❸ (Building (OBJECT ?building)(spatialCoordinates ?building_hasSpatialCoordinates))
  (SpatialCoordinates (OBJECT ?spatialCoord&:(eq ?spatialCoord ?building_hasSpatialCoordinates)))

❹ (GeoLocation (OBJECT ?locationPerson) (location ?locationPerson_coordinates))
  (GeoLocation (OBJECT ?locationBuilding) (location ?locationBuilding_coordinates))
  (SpatialCoordinates (OBJECT ?spatialCoord) (dimension ?spatialCoord_dimension))
  (test (call context_control.SpatialDimension Containment ?locationPerson_coordinates
                ?locationBuilding_coordinates ?spatialCoord_dimension))

❺ (Building (OBJECT ?building)(ID ?ID&:( eq (?ID "Zilverling"))))

  (not (SituationContained (OBJECT ?st)(person ?person) (building ?building) (finaltime nil)))
  =>
   (bind ?SituationContained (new situation_control.SituationContained ?person ?building))
   (definstance SituationContained ?SituationContained)
)
```

Appendix B

# ECA-DL Lifetime Constraints

The activity models presented in this appendix specify the execution cycle of an ECA-DL rule for the various lifetime constraints. *Figure B-1* depicts the execution cycle of an ECA-DL rule for the lifetime *once*. In this case, actions are invoked just once, and the rule is further deactivated.

*Figure B-1* ECA rule execution cycle for the lifetime *once*

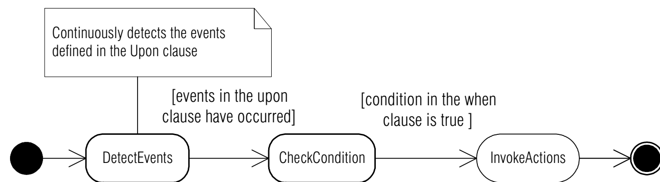



*Figure B-2* depicts the execution cycle of an ECA-DL rule for the lifetime *<n> times*. In this case, a counter is defined and incremented every time an action is invoked. When an action is invoked more than n times, the rule is deactivated.

*Figure B-2* ECA rule execution cycle for the lifetime *<n> times*

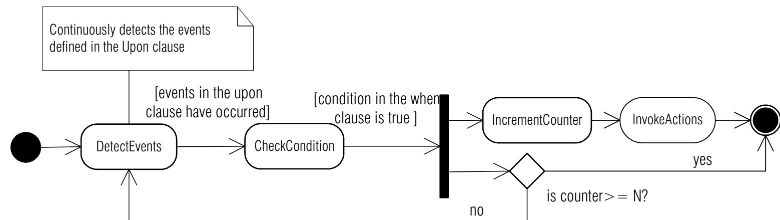



*Figure B-3* depicts the execution cycle of an ECA-DL rule for the lifetime *from <start> to <end>*. In this case, two checking points are necessary, namely one that detects whether it is time to start detecting events, and the other that checks whether it is already past the *<end>* period. If it is passed the *<end>* period, the rule is deactivated. Otherwise, the action is invoked and incoming events should be detected.

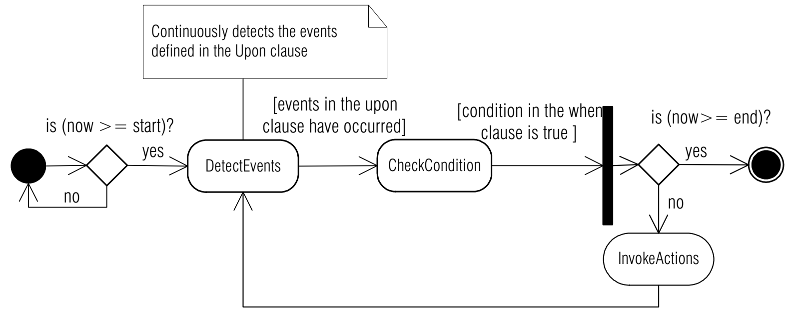



*Figure B-3* ECA rule execution cycle for the lifetime *from <start> to <end>*

*Figure B-4* depicts the execution cycle of an ECA-DL rule for the lifetime *to <end>*. In this case, one checking points is necessary, namely to detect whether the current time is past the *<end>* period. If it is passed the *<end>* period, the rule is deactivated. Otherwise, the action is invoked and incoming events should be detected.

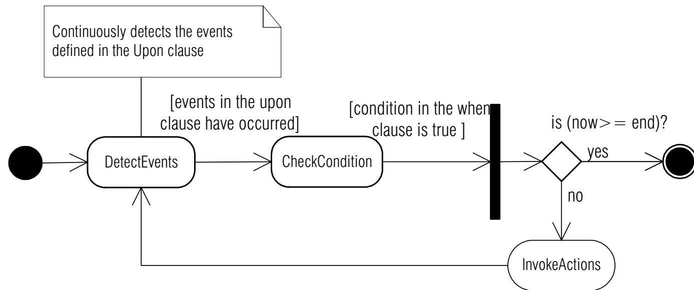



*Figure B-4* ECA rule execution cycle for the lifetime *to <end>*

Appendix C

# Mappings from ECA-DL to Jess

In this appendix we provide details of the mappings between ECA-DL expressions to Jess expressions. In addition, this appendix also presents examples of Jess rules, which are generated from ECA-DL rules using our mapping framework.

### *Term*

*Table C-1* provides examples of mappings from ECA-DL *terms* to the Jess language. These types of mappings have been extensively discussed Appendix A.

*Table C-1* Mapping Terms to Jess

| Term | ECA-DL expression | Jess expression |
|---|---|---|
| Entity | EntityType.id1 | (EntityType (OBJECT ?entity) (id id1)) |
| Reference Variable | variable | (VariableType (OBJECT ?variable)) |
| EntityContext | EntitytType.id1.object1.datatype.pdatatype | (EntityType (OBJECT ?entity) (id id1) (object1 ?object1))<br>(ObjectType1 (OBJECT ?object1) (datatype ?datatype))<br>(DataType (OBJECT ?datatype) (pdatatype ?pdatatype) |
| | variable.object1.datatype.pdatatype | (VariableType (OBJECT ?variable) (object1 ?object1))<br>ObjectType1 (OBJECT ?object1) (datatype ?datatype))<br>(DataType (OBJECT ?datatype) (pdatatype ?pdatatype) |
| EntityAttribute | EntitytType.id1.pdatatype | (EntityType (OBJECT ?entity) (id id1) (pdatatype ?pdatatype)) |
| | EntitytType.id1.datatype.pdatatype | (EntityType (OBJECT ?entity) (id id1) (datatype ?datatype))<br>(DataType (OBJECT ?datatype) (pdatatype ?pdatatype)) |
| Function | Function (EntityType.id1) | (EntityType (OBJECT ?entity) (id id1))<br>(test (call SomeClass Function ?entity)) |
| | Function (variable) | (test (call SomeClass Function ?variable)) |
| | Function (literal) | (test (call SomeClass Function literal)) |

| | | |
|---|---|---|
| | Function (variable.object1.datatype.pdatatype) | (EntityType (OBJECT ?entity) (id id1) (object1 ?object1))<br>(ObjectType1 (OBJECT ?object1) (datatype ?datatype))<br>(DataType (OBJECT ?datatype) (pdatatype ?pdatatype)<br>(test (call SomeClass Function ?pdatatype)) |
| ContextSituation | SituationType (EntityType.id1) | (EntityType (OBJECT ?entity) (id id1))<br>(SituationType (OBJECT ?situation) (entitytype ?entity)) |
| | SituationType (EntityType1.id1, EntityType2.id2 ) | (EntityType1 (OBJECT ?entity1) (id id1))<br>(EntityType2 (OBJECT ?entity2) (id id2))<br>(SituationType (OBJECT ?situation) (entitytype1 ?entity1) (entitytype2 ?entity2)) |
| | SituationType (variable) | (VariableType (OBJECT ?variable))<br>(SituationType (OBJECT ?situation) (variabletype ?variable)) |
| EntityCollection | EntityType.* | (EntityType (OBJECT ?entity)) |

***Binary and unary expressions***

We use the Jess equality operator eq (expr expr*) to test equality between objects and strings, which returns TRUE if the first argument is equal in type and value to all subsequent arguments. Similar functions (eq* or =) may be used to check equality for numeric values. Again, the combination of symbols &: is used to compare a slot values. The symbols &: reads "such that" in Jess. So, the code (…?object$_2$&:(eq (?object$_2$ ?object$_1$))) reads "object2 such that object2 is equal to object1". Similary, for numeric values, the expression (…?number$_1$&:(>(?number$_1$ ?number$_2$)) reads "number1 such that number1 is greater than number2.

When Boolean expression are used as parameters of functions in ECA-DL, such as Function (operand1 and operand2), these expressions are mapped to expressions using the Boolean operators and, or, not in Jess. Consider for example, the following ECA-DL expression, which invokes an action, which received two arguments, namely a unique identifier of the person, and a Boolean value which is true when this person's age is between 21 and 65 years old: Action (variable.id, variable.age > 21 or variable.age >65). This variable refers to each person of a collection of entities of type Person. This would be mapped to the following Jess expression:

```
(EntityType (OBJECT ?variable) (id ?id) (age ?age))
…
=>
(call NameClass Action ?id (or (>?age 21)  (> ?age 65)))
```

The parameters of a function should be always resolved before the function invocation expression, and in the LHS of a rule. *Table C-2* depicts some

examples of how binary expressions in ECA-DL can be mapped to binary expressions in Jess.

*Table C-2* Examples of mapping of binary expressions from ECA-DL to Jess

| ECA-DL expression | Jess expression |
|---|---|
| Situation (Entity.id1, Entity.id2) and Situation (variable) | (EntityType1 (OBJECT ?entity1) (id id1))<br>(EntityType2 (OBJECT ?entity2) (id id2))<br>(EntityType (OBJECT ?variable))<br>(SituationType (OBJECT ?situation) (entitytype1 ?entity1)<br>(entitytype2 ?entity2))<br>(SituationType (OBJECT ?situation) (entitytype ?variable)) |
| (EntitytType.id1.pdatatype1 > number) and (EntitytType.id1.object1.datatype.pdatatype2) | (EntityType (OBJECT ?entity) (id id1) (pdatatype1<br>?pdatatype1&:(> ?pdatatype1 number)))<br>(EntityType (OBJECT ?entity) (id id1) (object1 ?object1))<br>(ObjectType1 (OBJECT ?object1) (datatype ?datatype))<br>(DataType (OBJECT ?datatype) (pdatatype2 ?pdatatype2)) |
| (EntitytType.id1.object1.datatype.pdatatype > number) and (EntitytType.id2.object2.datatype2.pdatatype2 < number) | (EntityType (OBJECT ?entity) (id id1) (object1 ?object1))<br>(ObjectType1 (OBJECT ?object1) (datatype1 ?datatype1))<br>(DataType1 (OBJECT ?datatype1) (pdatatype1<br>?pdatatype1&: (>pdatatype1 number)))<br>(EntityType (OBJECT ?entity) (id id2) (object2 ?object2))<br>(ObjectType2 (OBJECT ?object2) (datatype2 ?datatype2))<br>(DataType2 (OBJECT ?datatype2) (pdatatype2<br>?pdatatype2&: (< pdatatype2 number))) |
| (not (EntitytType.id1.object1.datatype.pdatatype = number)) | (not (and ((EntityType (OBJECT ?entity) (id id1) (object1<br>?object1))<br>(ObjectType1 (OBJECT ?object1) (datatype<br>?datatype))<br>(DataType (OBJECT ?datatype) (pdatatype<br>?pdatatype&: (= pdatatype number))))) |
| Function (variable) or Function (Entity.id1) | (EntityType (OBJECT ?variable))<br>(EntityType (OBJECT ?entity) (id id1))<br>( or (test (call SomeClass Function ?variable))<br>(test (call SomeClass Function ?entity))<br>) |
| Function (not EntitytType.id1.pdatatype) | (EntityType (OBJECT ?entity) (id id1) (pdatatype<br>?pdatatype))<br>(test (call SomeClass Function (not ?pdatatype))) |

### *Mapping examples*

Consider the examples of ECA-DL rule specifications depicted in 7.2.6. For each of these rules, we partially depict in *Table C-3* the Jess rules that are generated. In addition to these rule, the following structures may be

also generated: a detection window rule; defqueries; a rule to update scope clause.

*Table C-3* Examples of mapping of ECA-DL rules

| ECA-DL rule | Jess rule 1 | Jess rule 2 |
|---|---|---|
| ECA Rule1:<br>Upon EpilepticAlarm<br>(Patient.John)<br>When SituationDriving<br>(Patient.John)<br>Do SendSms (Patient.John,<br>"John, you may have an epileptic seizure, please stop the car") | ev1:<br>EpilepticAlarm(Patient.John);<br>(defrule Jess1Rule1<br>(?eventfact <-<br>(EpilepticAlarm (id ev1)<br>(rule rule1)))<br>(not (and<br>(Patient (OBJECT ?patient)<br>(id "John"))<br>(SituationDriving<br>(person ?patient))))<br>=><br>(retract ?eventfact)) | (defrule Jess2Rule1<br>(?eventfact <-<br>(EpilepticAlarm (id ev1)<br>(rule rule1)))<br>(Patient (OBJECT ?patient)<br>(id "John"))<br>(SituationDriving<br>(person ?patient))<br>=><br>(retract ?eventfact)<br>(call SomeClass SendSms "John" "John, you may have an epileptic seizure, please stop the car")) |
| ECA Rule2:<br>Scope<br>(select<br>(Patient.*; patient;<br>patient.type = "epileptic"<br>and<br>patient.hasCivilLocation.city<br>= "Enschede") ; p)<br>{<br>Upon EpilepticAlarm (p)<br>When SituationDriving (p)<br>Do SendSms (p, "You may have an epileptic seizure, please stop the car")<br>} | ev1: EpilepticAlarm(P);<br>(defrule Jess1Rule2<br>(Patient (OBJECT ?p)<br>(id ?scopeid) (type "epileptic")<br>(hasCivilLocation<br>?hasCivilLocation))<br>(CivilLocation (OBJECT<br>?hasCivilLocation)<br>(city "Enschede"))<br>?eventfact <- (EpilepticAlarm<br>(eventID ev1) (ruleID rule2) (patientID ?scopeid))<br>(not (SituationDriving (person ?scopeid)))<br>=><br>(retract ?eventfact)) | (defrule Jess2Rule2<br>(Patient (OBJECT ?p)<br>(id ?scopeid) (type "epileptic")<br>(hasCivilLocation<br>?hasCivilLocation))<br>(CivilLocation (OBJECT<br>?hasCivilLocation)<br>(city "Enschede"))<br>?eventfact <- (EpilepticAlarm<br>(eventID ev1) (ruleID rule2) (patientID ?scopeid))<br>(SituationDriving (person ?scopeid))<br>=><br>(retract ?eventfact)<br>(call SomeClass SendSms ?scopeid "You may have an epileptic seizure, please stop the car")) |

*Table C-4* depicts some examples of mapping from ECA-DL to Jess for rules without a when clause.

*Table C-4* Examples of mapping of ECA-DL rules without when clause

| ECA-DL rule | Jess rule |
|---|---|
| ECA Rule3:<br>Scope (select (Patient.*; patient; patient.type = “diabetic”) ; p)<br>{<br>Upon HighSugarAlarm (p)<br>Do SendSms (p, “You have high sugar levels”)<br>} | ev1: HighSugarAlarm (P);<br>(defrule JessRule3<br>(Patient (OBJECT ?p) (id ?scopeid)<br>(type “diabetic”)<br>?eventfact <- (HighSugarAlarm<br>(eventID ev1) (ruleID rule3)<br>(patient ?scopeid))<br>=><br>(retract ?eventfact)<br>(call SomeClass SendSms ?scopeid “You have high sugar levels”)) |
| ECA Rule4:<br>Scope (Select (Policeman.*; policeman; policemen.hasActivity.value = ‘working’; p1)<br>{<br>Upon OnEvery (5)<br>Do NotifyApp (application-address,<br>List (p1.id, select(policeman.*, p2;<br>SituationWithinRange (p1,p2) and<br>p2.hasActivity.value = ‘working’)))<br>} | (Policemen (OBJECT ?p1) (id ?id1)<br>(hasActivity ?hasActivity))<br>(HasActivity (OBJECT ?hasActivity)<br>(value “working”))<br>?eventfact <- (OnEvery (eventID ev1) (ruleID rule4))<br>=><br>(retract ?eventfact)<br>(call SomeClass NotifyApp “application-address”<br>(call SomeClass List ?id1 (run-query getPolicemenWithinRange ?p1))) |

Appendix D

# Case Study Jess Rules

This appendix presents the Jess rules that correspond to the situation specifications and ECA-DL rules of the case studies discussed in Chapter 8.

***SituationCaregiverAvailable rules***

*Figure D-1* depicts the EnterTrue and EnterFalse Jess rules that are generated from the situation SituationCaregiverAvailable specification (*Figure 8-3*). The EnterTrue rule checks whether the OCL invariant holds and there is no current situation fact. If so, it creates an instance of situation SituationCaregiverAvailable, and invokes the notifySubscribers method, so that the SituationCaregiverAvailableContextManager can inform all the subscribers (in this case the healthcare controller) of that particular "EnterTrue" situation event. The EnterFalse rule checks whether this OCL invariant no longer holds and there is a current situation fact. If so, it deactivates the current situation fact, invokes the notifySubscribers method, so that the SituationCaregiverAvailableContextManager can inform the healthcare controller of that particular EnterFalse situation event.

*Figure D-1* Jess rules derived from Situation CaregiverAvailable specifications

```
;enter_true
(defrule enter_true__situationcaregiveravailable
    (Caregiver (OBJECT ?caregiver) (identity ?caregiverId) (hasCaregiverStatus ?status))
    (or (CaregiverStatus (OBJECT ?status) (status_value ?statusvalue&: (eq ?statusvalue 0)))
        (CaregiverStatus (OBJECT ?status) (status_value ?statusvalue&: (eq ?statusvalue 2))))
    (not (SituationCaregiverAvailable (care_giver ?caregiver) (care_giver_status ?status)))
    =>
    (bind ?SituationCaregiverAvailable (new situation_control.SituationCaregiverAvailable ?caregiver))
    (definstance SituationCaregiverAvailable ? SituationCaregiverAvailable)
    (bind ?initialtime (call ?SituationCaregiverAvailable getStarttime))
    (bind ?finaltime (call ?SituationCaregiverAvailable getFinaltime))
    (call SituationAvailableContextManager notifySubscribers "EnterTrue" ?caregiverId ?initialtime ?finaltime))

;enter_false
(defrule enter_false_situationcaregiveravailable
    (Caregiver(OBJECT ?caregiver) (identity ?caregiverId)(hasCaregiverStatus ?status))
    (not (or (CaregiverStatus (OBJECT ?status) (status_value ?statusvalue&: (eq ?statusvalue 0)))
            (CaregiverStatus (OBJECT ?status) (status_value ?statusvalue&: (eq ?statusvalue 2)))))
    (SituationCaregiverAvailable(OBJECT ?situation) (care_giver ?caregiver) (care_giver_status ?status))
    =>
    (call ?situation deactivate)
    (bind ?initialtime (call ?situation getStarttime))
    (bind ?finaltime (call ?situation getFinaltime))
    (call SituationAvailableContextManager notifySubscribers "EnterFalse" ?caregiverId ?initialtime ?finaltime))
```

### *SituationCaregiverWithinRange rules*

*Figure D-2* depicts the EnterTrue and EnterFalse Jess rules that are derived from the situation SituationCaregiverWithinRange specification (*Figure 8-4*). The EnterTrue rule checks whether the OCL invariant holds and there is no current situation fact. If so, it creates an instance of situation SituationWithinRange, and invokes the notifySubscribers method, so that the SituationWithinRangeContextManager can inform all the subscribers (in this case the healthcare controller) of that particular "EnterTrue" situation event. The EnterFalse rule checks whether this OCL invariant no longer holds and there is a current situation fact. If so, it deactivates the current situation fact, invokes the notifySubscribers method, so that the SituationWithinRangeContextManager can inform the healthcare controller of that particular EnterFalse situation event.

*Figure D-2* Jess rules derived from Situation CaregiverWithinRange specifications

```
;enter_true
(defrule enter_true_situationcaregiverwithinrange
    (EpilepticPatient (OBJECT ?patient) (identity ?patientId) (geoLocation ?patientLocation))
    (Caregiver (OBJECT ?caregiver&: (?caregiver isCaregiverOf ?caregiver ?patient)) (identity ?caregiverId) (geoLocation ?caregiverLocation))
    (GeoLocation (OBJECT ?patientLocation) (location ?location1))
    (GeoLocation (OBJECT ?caregiverLocation) (location ?location2))
    (test (< (call ?location1 Distance ?location1 ?location2 ) 100.0))
    (not (SituationCaregiverWithinRange (patient ?patient) (caregiver ?caregiver) (finaltime nil))
    =>
    (bind ?SituationCaregiverWithinRange (new situation_control.SituationCaregiverWithinRange ?patient ?caregiver))
    (definstance SituationCaregiverWithinRange ? SituationCaregiverWithinRange)
    (bind ?initialtime (call ?SituationCaregiverWithinRange getStarttime))
    (bind ?finaltime (call ?SituationCaregiverWithinRange getFinaltime))
    (call SituationCaregiverWithinRangeContextManager notifySubscribers "EnterTrue" ?patientId ?caregiverId ?initialtime ?finaltime))

;enter_false
(defrule enter_false_situationcaregiverwithinrange
    (EpilepticPatient (OBJECT ?patient) (identity ?patientId) (geoLocation ?patientLocation))
    (Caregiver (OBJECT ?caregiver&: (?caregiver isCaregiverOf ?caregiver ?patient)) (identity ?caregiverId) (geoLocation ?caregiverLocation))
    (GeoLocation (OBJECT ?patientLocation) (location ?location1))
    (GeoLocation (OBJECT ?caregiverLocation) (location ?location2))
    (not (test (< (call ?location1 Distance ?location1 ?location2 ) 100.0)))
    (SituationCaregiverWithinRange (OBJECT ?situation) (patient ?patient) (caregiver ?caregiver) (finaltime nil))
    =>
    (call ?situation deactivate)
    (bind ?initialtime (call ?situation getStarttime))
    (bind ?finaltime (call ?situation getFinaltime))
    (call SituationCaregiverWithinRangeContextManager notifySubscribers "EnterFalse" ?patientId ?caregiverId ?initialtime ?finaltime))
```

### *Jess rules for ECA-DL rules of the healthcare application*

The ECARule1 derived two rules, one that only consumes the event in case the when clause is false, and the other rule that consumes the event and invokes the action. We consider in the healthcare application a default detection window interval of one hour.

*Figure D-3* Jess rules derived from ECARule1

```
(defrule ECARule1_1
    ?event <-(EpilepticAlarm (idevent "ev1") (idrule "rl1") (patientID ?patientId) (timestamp ?t))
    (EpilepticPatient (OBJECT ?patient)(identity ?patientId) (hazardousActivity ?ha))
    (not (HazardousActivity (OBJECT ?ha) (hazardousvalue ?value&:(= ?value TRUE))))
    =>
    (retract ?event))

(defrule ECARule1_2
    ?event <-(EpilepticAlarm (idevent "ev1") (idrule "rl1") (patientID ?patientId) (timestamp ?t))
    (EpilepticPatient (OBJECT ?patient) (identity ?patientId) (hazardousActivity ?ha))
    (HazardousActivity (OBJECT ?ha) (hazardousvalue ?value&: (= ?value TRUE)))
    =>
    (retract ?event)
    (call HealthcareActionResolver notifyPatientApplication ?patientId))
```

The ECARule2 rule derives a Jess rule and a defquery. Only one Jess rule is derived since there is no when clause defined in this ECA-DL rule. This rule consumes the event and invokes the action.

*Figure D-4* Jess rules derived from ECARule2

```
(defquery SelectCaregiversECARule2
    (declare (variables ?patient))
    (Caregiver (OBJECT ?caregiver&:(?caregiver isCaregiverOf ?caregiver ?patient)) (identity ?idcaregiver))
    (SituationCaregiverAvailable (caregiver ?caregiver) (finaltime nil))
    (SituationCaregiverWithinRange (caregiver ?caregiver) (patient ?patient) (finaltime nil)))

(defrule ECARule2
    ?event <- (EpilepticAlarm (idevent "ev1") (idrule "rl2") (patientID ?patientId) (timestamp ?t))
    (EpilepticPatient (OBJECT ?patient)(identity ?patientId) (geoLocation ?geolocation))
    (GeoLocation (OBJECT ?geoLocation) (location ?coordinates))
    =>
    (retract ?event)
    (bind ?query_result (run-query* SelectCaregiversECARule2 ?patient))
    (bind ?result (call Util List ?query_result))
    (call HealthcareActionResolver notifyCaregiversApplications ?patientId ?coordinates ?result))
```

The ECARule3 rule derives a Jess rule and a defquery. Only one Jess rule is derived since there is no when clause defined in this ECA-DL rule. This rule consumes the event and invokes the action.

*Figure D-5* Jess rules derived for ECARule3

```
(defquery SelectCaregiversECARule3
   (declare (variables ?patient ?caregiver))
   (Caregiver (OBJECT ?care&: (and (?care isCaregiverOf ?care ?patient) (neq ?care ?caregiver))) (identity ?idcaregiver))
   (SituationCaregiverAvailable (caregiver ?care) (finaltime nil))
   (SituationCaregiverWithinRange (caregiver ?care) (patient ?patient)(finaltime nil)))

(defrule ECARule3
    ?event1 <- (EpilepticAlarm (idevent "ev1") (idrule "rl3") (timestamp ?t1) (patientID ?patientId))
    ?event2 <- (AcceptRequestEvent (idevent "ev2") (idrule "rl3") (patientID ?patientId) (careGiverID ?caregiverId)
              (timestamp ?t2&:(> ?t2 ?t1)))
    (EpilepticPatient (OBJECT ?patient) (identity ?patientId))
    (Caregiver (OBJECT ?caregiver) (identity ?caregiverId))
    =>
   (retract ?event1)
   (retract ?event2)
   (bind ?query_result (run-query* SelectCaregiversECARule3 ?patient ?caregiver))
   (bind ?result (call Util List ?query_result))
   (call HealthcareActionResolver notifyAcceptanceCaregivers ?result ?patientId ?caregiverId))
```

The ECARule4 rule derives a Jess rule and a defquery. Only one Jess is derived since there is no when clause defined in this ECA-DL rule. This rule consumes the event and invokes the action.

*Figure D-6* Jess rules derived for ECARule4

```
(defquery SelectHealthProfessionalsECARule4
    (declare (variables ?patientID))
    (EpilepticPatient (OBJECT ?patient) (identity ?patientID))
    (HealthProfessional (OBJECT ?healthprofessional) (identity ?healthprofessionalID))
    (test (call ? healthprofessional isHealthProfessionalOf ? healthprofessional ?patient)))

(defrule ECARule4
    ?event <- (EpilepticAlarm (idevent "ev1") (idrule "rl4") (patientID ?patientID) (timestamp ?t))
    =>
    (retract ?event)
    (bind ?query_result (run-query* SelectHealthProfessionalsECARule4 ?patientID))
    (bind ?result (call Util List ?query_result))
    (call HealthcareActionResolver logEpilepticAlarm ?patientId ?result ))
```

***Jess rules for ECA-DL rules of the policy management application***

The ECARule5 rule derives a Jess rule and a defquery.

*Figure D-7* Jess rules derived from ECARule5

```
(defquery SelectCaregiversECARule5
    (declare (variables ?patient))
    (Caregiver (OBJECT ?caregiver&:(?caregiver isCaregiverOf ?caregiver ?patient)) (identity ?idcaregiver))
    (SituationCaregiverAvailable (caregiver ?caregiver) (finaltime nil))
    (SituationCaregiverWithinRange (caregiver ?caregiver) (patient ?patient)(finaltime nil)))

(defrule ECARule5
    ?event <- (EpilepticAlarm (idevent "ev1") (idrule "rl5") (patientID ?patientId) (timestamp ?t))
    (EpilepticPatient (OBJECT ?patient)(identity ?patientId) (geoLocation ?geolocation))
    =>
    (retract ?event)
    (bind ?query_result (run-query* SelectCaregiversECARule5 ?patient))
    (bind ?result (call Util List ?query_result))
    (call PolicyManagementActionResolver grantAccessControlPolicyPatient ?patientId ?result)
    (call PolicyManagementActionResolver grantPolicyCaregiversStatus ?result ?patientId))
```

The ECARule6 rule derives a Jess rule and a defquery. Only one Jess rule is derived since there is no when clause defined in this ECA-DL rule. This rule consumes the event and invokes the action.

*Figure D-8* Jess rules derived for ECARule6

```
(defquery SelectCaregiversECARule6
   (declare (variables ?patient ?caregiver))
   (Caregiver (OBJECT ?care&: (and (?care isCaregiverOf ?care ?patient) (neq ?care ?caregiver))) (identity ?idcaregiver))
   (SituationCaregiverAvailable (caregiver ?care)(finaltime nil))
   (SituationCaregiverWithinRange (caregiver ?care) (patient ?patient)(finaltime nil)))

(defrule ECARule6
     ?event1 <- (EpilepticAlarm (idevent "ev1") (idrule "rl6") (timestamp ?t1) (patientID ?patientId))
     ?event2 <- (AcceptRequestEvent (idevent "ev2") (idrule "rl6") (patientID ?patientId) (careGiverID ?caregiverId)
                (timestamp ?t2&:(> ?t2 ?t1)))
     (EpilepticPatient (OBJECT ?patient) (identity ?patientId))
     (Caregiver (OBJECT ?caregiver) (identity ?caregiverId))
     =>
   (retract ?event1)
   (retract ?event2)
   (bind ?query_result (run-query* SelectCaregiversECARule6 ?patient ?caregiver))
   (bind ?result (call Util List ?query_result))
   (call PolicyManagementActionResolver denyAccessControlPolicyPatient ?patientId ?result)
   (call PolicyManagementActionResolver increaseTrustPolicyPatient ?patientId ?caregiverId))
```

The ECARule7 rule derives a Jess rule, which consumes the event and invokes the action.

*Figure D-9* Jess rules derived for ECARule7

```
(defrule ECARule7
     ?event1 <- (EpilepticAlarm (idevent "ev1") (idrule "rl7") (timestamp ?t1) (patientID ?patientId))
     ?event2 <- (AcceptRequestEvent (idevent "ev2") (idrule "rl7") (patientID ?patientId) (careGiverID ?caregiverId)
                (timestamp ?t2&:(> ?t2 ?t1)))
     (EpilepticPatient (OBJECT ?patient) (identity ?patientId))
     (Caregiver (OBJECT ?caregiver) (identity ?caregiverId))
     =>
   (retract ?event1)
   (retract ?event2)
   (bind ?query_result (run-query* SelectCaregiversECARule6 ?patient ?caregiver))
   (bind ?result (call Util List ?query_result))
   (call PolicyManagementActionResolver increaseTrustPolicyPatient ?patientId ?caregiverId))
```

The ECARule8 rule derives a Jess rule. Only one Jess is derived since there is no when clause defined in this ECA-DL rule. This rule consumes the event and invokes the action.

*Figure D-10* Jess rules derived for ECARule8

```
(defrule ECARule8
   ?event <- (RejectHelpRequest (idevent "ev3") (idrule "rl8") (careGiverID ?caregiverId)
                                                  (patientID ?patientId) (timestamp ?t))
   =>
   (retract ?event)
   (call PolicyManagementActionResolver decreaseTrustPolicyPatient ?patientId ?caregiverId))
```

# References


1. C. Alexander, *The Timeless Way of Building*, Oxford University Press, 1979.

2. J.P. Andrade Almeida, *Model-Driven Design of Distributed Applications*, Ph.D. Thesis in Computer Science, CTIT Ph.D.-Thesis Series, No. 06-85, Telematica Instituut Fundamental Research Series, No. 018 (TI/FRS/018), Enschede, The Netherlands, 2006.

3. ATLAS group, LINA & INRIA, ATL Starter's Guide. Version 0.1, December 2005. Available at: http://www.eclipse.org/m2m/atl/doc/ATL_Starter_Guide.pdf

4. ATLAS group, LINA & INRIA, ATL User Manual. Version 0.7, February 2006. Available at: http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf

5. J. Bardram, "Applications of Context-Aware Computing in Hospital work", Proc. of ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, 2004, pp. 1547-1579.

6. H. Batteram, et al., "AWARENESS Scope and Scenarios". AWARENESS Deliverable D1.1, 2004. Available at: http://awareness.freeband.nl

7. G. Biegel and V. Cahill, "A Framework for Developing Mobile, Context-Aware Applications", Proc. of the 2nd IEEE Conference on Pervasive Computing and Communications (Percom2004), Orlando USA, 2004.

8. L.O. Bonino da Silva Santos, M. van Sinderen, and L. Ferreira Pires, "Dynamic Service Discovery and Composition for Ubiquitous Networks Applications", Second Conference on Future Networking Technologies (CoNEXT 2006), Poster Track, Lisbon, Portugal, December 2006.

9. T. Buchholz, A. Küpper, and M. Schiffers, "Quality of context: What it is and why we need it". In Proceedings of the Workshop of the HP OpenView University Association 2003 (HPOVUA 2003), Geneva, 2003.

10. F. Buschmann, et al., “Pattern-Oriented software architecture: A System of Patterns”. John Wiley and Sons, New York, U.S.A. , 2001.
11. Web Services Business Process Execution Language v. 2.0, OASIS Committee Specification, Jan. 31, 2007.
12. D. Brickley and L. Miller. “FOAF vocabulary specification”, In RDFWeb Namespace Document. RDFWeb, xmlns.com, 2003.
13. J. Bauer, *Identification and Modeling of Contexts for Different Information Scenarios in Air Traffic*, Diplomarbeit¸ Technical University of Berlin, Mar. 2003.
14. F. Cabitza, M. Sarini, B. Dal Seno, “DJess - a context-sharing middleware to deploy distributed inference systems in pervasive computing domains”, Proc. Int’l Conference on Pervasive Services (ICPS '05), IEEE CS Press, 2005.
15. H. Chen, T. Finin, and A. Joshi, “An ontology for context-aware pervasive computing environments”, Knowledge Engineering Review, Special Issue on Ontologies for Distributed System, 2003.
16. H. Chen, *An Intelligent Broker Architecture for Pervasive Context-Aware Systems*, Ph.D Thesis, University of Maryland, Baltimore County, December 2004.
17. H. Chen, et al., “SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications”, Proc. International Conference on Mobile and Ubiquitous Systems: Networking and Services, August 2004.
18. G. Chen and D. Kotz, “A survey of context-aware mobile computing research”, Technical Report TR2000-381, Department of Computer Science, Dartmouth College, 2000.
19. G. Riley, Clips: A Tool for Building Expert Systems, CLIPS website. Available at: http://www.ghg.net/clips/CLIPS.html
20. Context-Aware, the MIT context-aware group. Available at: http://context.media.mit.edu/press
21. L. M. Daniele, *Towards a Rule-Based Approach for Context-Aware Applications*, M.Sc. thesis, Dept. Computer Science, University of Twente, Enschede, The Netherlands, 2006.
22. A. Dey, G. D. Abowd, and D. Salber, “A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications”, Human-Computer Interaction, 16:97–166, 2001.
23. A. Dey, “Context-aware computing: The CyberDesk Project”, Proc. of AAAI 1998 Spring Symposium Series on Intelligent Environments, Dagsthul, Germany, 1998.
24. A. Dey, “Supporting the construction of context-aware applications”, Presentation delivered at the Dagstuhl Seminar on Ubiquitous Computing, Dagsthul, Germany, 2001.

25. A. Dey and G. Abowd, "Towards a better understanding of context and context-awareness", Technical ReportGIT-GVU-99-22, Georgia Institute of Technology, 1999.

26. K. Devlin, *Logic and Information*, Cambridge University Press, 1995.

27. M.G. Davies and D.M. Thomson, "Introduction", in *Memory in context; context in memory* (G. M. Davies and D. M. Thomson, Eds.). Chichester, England: Wiley, 1998, pp. 1–10.

28. P. Dockhorn Costa, "Towards a Services Platform for Context-Aware Applications", M.Sc. thesis, Dept. Computer Science., University of Twente, Enschede, The Netherlands, 2003.

29. P. Dockhorn Costa, et al., "AWARENESS Services Infrastructure", AWARENESS Deliverable D2.1, 2004. Available at: http://awareness.freeband.nl

30. P. Dockhorn Costa, L. Ferreira Pires, M. van Sinderen, and J. Pereira Filho, "Towards a Service Platform for Mobile Context-Aware Applications", Proc. 1st Intl. Workshop on Ubiquitous Computing (IWUC 2004), Portugal, 2004, pp. 48-61.

31. P. Dockhorn Costa, L. Ferreira Pires, M. van Sinderen, and D. Rios, "Services Platforms for Context-Aware Applications", Proc. 2nd European Symp. on Ambient Intelligence (EUSAI 2004), The Netherlands, 2004, pp. 363-366.

32. P. Dockhorn Costa, J.P.A. Almeida, L. Ferreira Pires, G. Guizzardi, and M. van Sinderen, "Towards Conceptual Foundations for Context-Aware Applications", Proc. of the Third International Workshop on Modeling and Retrieval of Context (MRC06), Boston, USA, 2006.

33. P. Dockhorn Costa, L. Ferreira Pires, and M. van Sinderen, "Designing a Configurable Services Platform for Mobile Context-Aware Applications", Int'l Journal of Pervasive Computing and Communications (JPCC), 1(1):13-25, Troubador Publishing, 2005.

34. Eclipse Foundation, "Eclipse - an open development platform", Eclipse website. Available at: http://www.eclipse.org

35. SourceForge.net, Eclipse Metrics plugin website. Available at: http://sourceforge.net/projects/metrics

36. Eclipse Foundation, the ATLAS Transformation Language. Available at: http://www.eclipse.org/m2m/atl/

37. R. Etter, P. Dockhorn Costa and T. Broens, "A Rule-Based Approach Towards Context-Aware User Notification Services", Proc. of the IEEE Int'l Conference on Pervasive Services, Lyon, France, June 2006.

38. P. Eugster, P. Felber, R. Guerraoui and A.M. Kermarrec, " The Many Faces of Publish/Subscribe", ACM computing Surveys, Vol. 35(2), June 2003.

39. Sun Microsystems, Enterprise Java Beans technology. Available at: http://java.sun.com/products/ejb/

40. L. Ferreira Pires, *Architectural Notes: a framework for distributed systems development*, Ph.D. Thesis, University of Twente, The Netherlands, Sept. 1994.

41. Freeband Kennisimpuls, "AWARENESS project". The Netherlands, 2004. Available at: http://awareness.freeband.nl

42. E. Friedman-Hill, *Jess in Action*, Java Rule Based Systems, Manning Publications Co., 2003.

43. A. Fetzer, "Recontextualizing context", Proc. of Context Organiser workshop at ECCS '97. April 9–11, Manchester, UK, 1997.

44. C. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem", Artificial Intelligence 19, 1982, pp. 17-37.

45. F. Fuchs, I. Hochstatter, M. Krause and M. Berger, "A Meta–Model Approach to Context Information", Proc. of 2nd IEEE PerCom Workshop on Context Modeling and Reasoning (CoMoRea) (at 3rd IEEE International Conference on Pervasive Computing and Communication (PerCom 2005)), IEEE, Hawaii, USA, March, 2005.

46. Gentleware products, Poseidon for UML. Available at: http://www.gentleware.com/products.html

47. E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts, U.S.A., Addison Wesley, 1996.

48. P. Gray and D. Salber, "Modeling and Using Sensed Context Information in the design of Interactive Applications", Proc. of 8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI 01), Toronto, Canada, May 2001.

49. R. Guha, "Contexts: a formalization and some applications", Technical Report, Stanford University, Stanford, CA, 1992. Available at: http://www-formal.stanford.edu/guha/guha-thesis.ps

50. G. Guizzardi, H. Herre and G. Wagner, "On the General Ontological Foundations of Conceptual Modeling", Proc. of the 21st Int'l Conference on Conceptual Modeling (ER-2002), LNCS 2503, 2002.

51. G. Guizzardi, *Ontological Foundations for Structural Conceptual Models*, PhD Thesis, University of Twente, The Netherlands. TI-FRS No. 15, 2005.

52. G. Guizzardi, and G. Wagner, "Towards Ontological Foundations for Agent Modeling Concepts using UFO", Agent-Oriented Information Systems (AOIS), Lecture Notes on Artificial Intelligence (LNAI) 3508, Springer-Verlag, 2005.

53. X. Hang Wang, D. Qing Zhang, T. Gu, and H. Keng Pung, "Ontology-Based Context Modeling and Reasoning Using OWL", Proc. of the 2nd IEEE Conf.

on Pervasive Computing and Communications Workshop (PERCOMW04), USA, 2004.

54. C. Hesselman, A. Tokmakoff, P. Pawar, and S. Iacob, “Discovery and Composition of Services for Context-Aware Systems”, Proc. of the 1st IEEE European Conference on Smart Sensing and Context, Enschede, The Netherlands, October 2006.

55. B. Heller and H. Herre, “Ontological Categories in GOL”, Axiomathes 14:71-90 Kluwer Academic Publishers, 2004.

56. K. Henricksen and J. Indulska, “A software engineering framework for context-aware pervasive computing”, Proc. of the 2nd IEEE Conf. on Pervasive Computing and Communications (Percom2004), USA, IEEE CS Press, 2004.

57. K. Henricksen, *A framework for context-aware pervasive computing applications*, PhD thesis, School of Information Technology and Electrical Engineering, The University of Queensland, September 2003.

58. K. Henricksen, J. Indulska, T. McFadden, and S. Balasubramaniam, “Middleware for distributed context-aware systems”, Proc. of the Int’l Symposium on Distributed Objects and Applications (DOA), volume 3760 of Lecture Notes in Computer Science, Springer, 2005, pp. 846-863.

59. M. Hockenberry, R. Gens, and T. Selker, “User Centered Mapping: Theoretical and Practical Framework, MIT Whitepaper, 2005. Available at: http://placemap.mit.edu/papers/ucm.pdf

60. IBM Rational Software, Rational Rose modelling products. Available at: http://www-306.ibm.com/software/awdtools/developer/rose

61. ITU-T X.901 | ISO/IEC 10746-1 ODP Reference Model Part 1. Overview. 1995.

62. ISO/IEC, Open Distributed Processing Reference Model, Part 3: Architecture, IS 10746-3 | X.903.

63. Java Remote Method Invocation (Java RMI) website. Available at: http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

64. Sandia Labs, Jess: the Rule Engine for the Java Platform, Jess website. Available at: http://herzberg.ca.sandia.gov/jess/

65. SourceForge.net, JDrew: the Java Deductive Reasoning Engine for the Web, jDREW website. Available at: http://www.jdrew.org/jDREWebsite/jDREW.html

66. L. Kagal, T. Finin, and A. Joshi, “A Policy Based Approach to Security for the Semantic Web”, Proc. 2nd International Semantic Web Conference (ISWC2003), September 2003.

67. M. Kifer, G. Lausen, and J. Wu, "Logical foundations of object-oriented and framebased languages", ACM 42, 1995, pp. 741–834.

68. A. Kleppe and J. Warmer, "Wed Yourself to UML with the Power of Associations", DevX online magazine article, June 2005. Available at: http://www.devx.com/enterprise/Article/28528/0

69. C.H. Lee, L. Bonanni, J.H. Espinosa, H. Lieberman, and T. Selker, "Augmenting Kitchen Appliances with a Shared Context Using Knowledge about Daily Events", Proc. of Int'l Conference on Intelligent User Interfaces (IUI), Sydney, Australia, 2006.

70. D. B. Lenat and R. V. Guha, *Building Large Knowledge-Based Systems: Representation and Inference in the CycProject*. Addison-Wesley, February 1990.

71. The Mandarax project, Mandarax website. Available at: http://mandarax.sourceforge.net

72. C. Masolo, S. Borgo, A. Gangemi, N. Guarino, and A. Oltramari, "Ontology Library", WonderWeb deliverable D18, 2003.

73. J. McCarthy, "Notes on formalizing context", Proc. of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93), Morgan Kaufmann, Mountain View, CA, 1993. Available at: http://www-formal.stanford.edu/jmc/home.html

74. D.L. McGuinness and F. van Harmelen, "OWL web ontology language overview", 2004. Available at: http://www.w3.org/TR/owl-features

75. N. Maatjes, *Automated transformations from ECA rules to Jess*, M.Sc. thesis, Dept. Computer Science., University of Twente, Enschede, The Netherlands, 2007.

76. M. Mansouri-Samani and M. Sloman, "GEM: A Generalized Event Monitoring Language for Distributed Systems", IEE/IOP/BCS Distributed Systems Engineering . Journal., vol. 4, no. 2, June 1997.

77. E. Meeuwissen, et al., "Functional Architecture of the AWARENESS Infrastructure", AWARENESS Deliverable D1.3 (2005). Available at: http://awareness.freeband.nl

78. The MyHeart Consortium, MyHeart project webpage. Available at: http://www.extra.research.philips.com/euprojects/myheart/

79. Merriam-Webster, Inc., Merriam-Webster Online. Available at: http://m-w.com

80. J. Mylopoulos, "Conceptual modeling and Telos", in *Conceptual modeling, databases, and CASE* (P. Loucopoulos and R. Zicari, Eds), Wiley, pp. 49-68.

81. T. P. Moran and P. Dourish, "Introduction to This Special Issue on Context-Aware Computing", Special Issue of Human-Computer Interaction, Volume 16, IBM Almaden Research Center, University of California, Irvine, 2001.

82. K. Mulligan and B. Smith, A Relational theory of the Act. Topoi (5/2), 115-30, 1986.

83. R. Neisse, M. Wegdam, and M. van Sinderen, "Context-Aware Trust Domains", Proc. of the 1st European Conference on Smart Sensing and Context, Enschede, The Netherlands, 2006.

84. R. Neisse, M. Wegdam, P. Dockhorn Costa, and M. van Sinderen, "Context-Aware Management Domains", Proc. 1st Int'l Workshop on Combining Context with Trust, Security, and Privacy Moncton, Canada, Jul-2007.

85. N.A. Bradley and M.D. Dunlop, "Toward a Multidisciplinary Model of Context to Support Context-Aware Computing", Human Computer Interaction, 2005, Volume 20, pp. 403 446.

86. Object Management Group, "A Discussion of the Object Management Architecture", formal/00-06-41, January 1997.

87. Object Management Group, "CORBA Components (version 3.0)", formal/02-06-65, June 2002.

88. Object Management Group, "Trading Object Services Specification", Version 1.0. Available at: http://www.omg.org/docs/formal/00-06-27.pdf

89. Object Management Group, "Notification Service Specification", Version 1.1. Available at: http://www.omg.org/docs/formal/04-10-11.pdf

90. Object Management Group, "Unified Modelling Language: Object Constraint Language", version 2.0, ptc/03-10-04, 2003.

91. Object Management Group, "UML 2.0 Superstructure", ptc/03-08-02, 2003.

92. Object Management Group, "the Model Driven Architecture". Available at: http://www.omg.org/mda/

93. Object Management Group, MetaObject Facility. Available at: http://www.omg.org/mof/

94. Objecteerring software, Objeteering/UML. Available at http://www.objecteering.com/products.php

95. E. Ochs, " What a child language can contribute to pragmatics", in *Developmental pragmatics* (E. Ochs and B. B. Schriffin, Eds.), New York Academic, 1979, pp. 1–17.

96. Octopus: OCL Tool for Precise UML Specifications Website. Available at: http://www.klasse.nl/octopus/index.html

97. F. Perich. MoGATU BDI Ontology, 2004.

98. S. Powers. Practical RDF. O'Reilly & Associates, 2003.

99. P. Pawar and A. Tokmakoff, "Ontology-Based Context-Aware Service Discovery for Pervasive Environments", Proc. 1st IEEE International

Workshop on Services Integration in Pervasive Environments (SIPE 2006), Co-located with IEEE ICPS 2006, Lyon, France, June 2006.

100. F. Pan and J. R. Hobbs, "Time in OWL-S", Proc. of AAAI-04 Spring Symposium on SemanticWeb Services, Stanford University, California, 2004.

101. Parlay Group, "Parlay X Web Services White Paper", 2002. Available at: http://www.parlay.org/ about/parlay_x/ParlayX-WhitePaper-1.0.pdf

102. M.P. Papazoglou and D. Georgakopoulos, "Service-oriented computing", Communications of the ACM 46(10): 25-28, October 2003.

103. D. Preuveneers, et al., "Towards an extensible context ontology for ambient intelligence", Proc. Second European Symposium on Ambient Intelligence (EUSAI 2004), Eindhoven, the Netherlands, 2004.

104. S. Pokraev, J. Koolwaaij, M. van Setten, T. Broens, P. Dockhorn Costa, M. Wibbels, P. Ebben, and P. Strating, "Service Platform for Rapid Development and Deployment of Context-Aware, Mobile Applications", Proc. of International Conference on Webservices (ICWS'05), Orlando, Florida, USA, 2005.

105. M. Román, C.K. Hess, R. Cerqueira, A. Ranganathan, R.H. Campbell, and K. Nahrstedt, "Gaia: A Middleware Infrastructure to Enable Active Spaces", IEEE Pervasive Computing, Oct-Dec 2002, pp. 74-83.

106. D. Quartel, *Action relations Basic design concepts for behaviour modelling and refinement*, Ph.D. thesis, University of Twente, The Netherlands, Feb. 1998.

107. D. Rapela, et al., MDA modeling and application principles, MODA-TEL Consortium, Deliverable 3.3, 2004. Available at: http://www.modatel.org/public/deliverables/D3.3.htm

108. P. Ross, "Managing Care Through the Air", IEEE Spectrum, December 2004, pp. 26-31.

109. D. A. Randell, Z. Cui, and A. Cohn, "A spatial logic based on regions and connection", Proc. of the Third International Conference, Morgan Kaufmann, San Mateo, California, 1992, pp. 26-31.

110. Roessingh Research and Development website. Available at: http://www.rrd.nl/www/indexa.html

111. K. Sheikh, M. Wegdam and M.J. van Sinderen, "Middleware Support for Quality of Context in Pervasive Context-Aware Systems", IEEE Percom workshop proceedings, part of the Proceedings of the Fifth Annual IEEE International Conference on Pervasive Computing and Communications, 2007.

112. S. Smith, "Environmental context-dependent memory", In *Memory in context; context in memory* (G. M. Davies and D. M. Thomson, Eds.). Chichester, England: Wiley, 1988, pp. 13–34.

113. B.N. Schilit, N. Adams, and R. Want, “Context-aware computing applications”, Proc. Workshop on Mobile Computing Systems and Applications, December 1994.

114. M.J. van Sinderen, A.T. van Halteren, M. Wegdam, H.B. Meeuwissen, and E.H. Eertink, “Supporting Context-aware Mobile Applications: an Infrastructure Approach”, IEEE Communications Magazine, 44 (9). Sept 2006, pp. 96-104.

115. T. Strang and C. Linnhof-Popien, “A context modeling survey”, Proc. 1st International Workshop on Advanced Context Modelling, Reasoning and Management, Nottingham (2004) 34-41.

116. Tom Tom, the Navigatiesystemen, Tom Tom online. Available at: http://www.tomtom.com.

117. T. Strang, C. Linnhoff-Popien, and K. Frank, "CoOL: A Context Ontology Language to enable Contextual Interoperability," Proc. 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS2003), 2003.

118. S.S. Yau, F. Karim, Y. Wang, B. Wang, S.K.S. Gupta, “Reconfigurable Context-Sensitive Middleware for Pervasive Computing”, IEEE Pervasive Computing, vol. 01, no. 3, Jul-Sept 2002, pp. 33-40.

119. Jie Yang, Weiyi Yang, Denecke, M., Waibel, A. (1999). Smart sight: a tourist assistant system. 3rd International Symposium on Wearable Computers, San Francisco, California, 18-19 October, 1999, pp. 73-78.

120. Universal Description, Discovery and Integration (UDDI) project: UDDI: Specifications. Available at: http://www.uddi.org/specification.html.

121. A. Vallecillo, RM-ODP: The ISO Reference Model for Open Distributed Processing, DINTEL Edition on Software Engineering. No. 3. pp. 69-99. March 2001.

122. R. Want, A. Hopper, V. Falcao, and J. Gibbons, “The Active Badge Location System”, ACM Transactions on Information Systems 10(1), 1992, pp. 91-102.

123. R. Want, et al., “An Overview of the PARCTAB Ubiquitous Computing Environment”, EE Personal Communications, vol 2, no 6, Dec 1995, pp. 28-43.

124. M. Weiser, “The Computer for the 21st Century”, Scientific American, 265(3):66–75, September 1991.

125. World Wide Web Consortium: Web Services Architecture, 2004. Available at: http://www.w3.org/TR/ws-arch/

126. Wikipedia, the Free Encyclopedia. Available at: http://en.wikipedia.org/wiki

127. A. Zaslavsky, "Mobile Agents: Can They Assist with Context Awareness?", School of Computer Science and Software Engineering, Monash University, Australia.

128. T. Ziemke, "Embodiment of context", Proc. of ECCS, Manchester, UK, April 1997.

129. C. Zetie, "Market overview - The emerging context-aware software market", 2002. Available at: http://ww.forrester.com/Research/LegacyIT/Excerpt/0,7208,25595,00.html

# Resumo

Sensibilidade ao contexto do usuário tornou-se uma importante e desejável característica em aplicações ubíquas. Esta característica permite que as aplicações usem informações do ambiente (contexto) para customizar seus serviços de acordo com a situação e as necessidades atuais de seus usuários.

Esta tese tem como objetivo a proposta de uma solução integrada para o desenvolvimento de aplicações sensíveis ao contexto. O objetivo principal é facilitar o desenvolvimento destas aplicações com ênfase em dois aspectos: (i) na modelagem do contexto e das situações na qual usuários podem se encontrar; e (ii) no apoio infraestrutural a aplicações por meio de uma plataforma de gerenciamento de contexto e de detecção de situações.

Nossa proposta de modelagem de contexto oferece aos desenvolvedores de aplicações conceitos básicos que podem ser extendidos e customizados com requisitos específicos de aplicação. A plataforma de gerenciamento de contexto permite que funcionalidades específicas das aplicações sejam delegadas para a plataforma, reduzindo os esforços e o tempo de desenvolvimento da aplicação, e por consequência, reduzindo os custos de desenvolvimento. Isto permite que os desenvolvedores de aplicação se concentrem nos aspectos principais dos seus negócios ao invés de serem distraídos com detalhes de realização da aplicação.

À medida que aplicações tornam-se mais complexas, há uma necessidade crescente de abstrações de modelagem que são apropriadas para: (i) caracterizar o universo de discurso das aplicações; (ii) promover o entendimento, resolução de problemas e a comunicação entre os “stakeholders” envolvidos no desenvolvimento da aplicação; e (iii) respresentar o contexto sem ambigüidades. Com o objetivo de satisfazer os requisitos de modelagem de contexto, nós definimos um conjunto de abstrações conceituais que são baseados em teorias de modelagem conceitual estabelecidas na literatura.

Na abordagem proposta nesta tese, aplicações sensíveis ao contexto usam e manipulam informações contextuais para detectar situações de alto

nível, que são usadas para adaptar o comportamento das aplicações. Nós propomos uma abordagem orientada a modelos para a especificação de situações e introduzimos uma abordagem baseada em regras para implementar a detecção de situações.

Situações são especificadas usando uma combinação de diagramas de classe UML, e restrições OCL. As situações podem ser compostas a partir de tipos elementares de contexto.

Este trabalho discute ainda como gerenciar a distribuição e como explorá-la beneficialmente na manipulação de contexto e na detecção de situações de modo distribuído.

Finalmente, nós propomos um mecanismo para facilitar a configuração e a execução de comportamentos da aplicação na plataforma de gerenciamento de contexto, em tempo de execução da plataforma. Este mecanismo é baseado em uma máquina de regras que autonomamente coleta contexto e valores das situações originadas de componentes processadores de contexto, que estão distribuídos. Esta máquina aceita especificações de comportamento de aplicação escritos em ECA-DL. ECA-DL é uma linguagem específica de domínio desenvolvida no escopo desta tese com o propósito de especificar os comportamentos das aplicações sensíveis ao contexto.

# Publications by the Author

During the development of this thesis, the author has published various parts of this work in the following papers (listed in reverse chronological order):

- R. Neisse, M. Wegdam, P. Dockhorn Costa, M. van Sinderen, "Context-Aware Management Domains", Proc. of the First International Workshop on Combining Context with Trust, Security, and Privacy, Moncton, Canada, Jul-2007.
- L. Daniele, P. Dockhorn Costa, L. Ferreira Pires, "Towards a Rule-Based Approach for Context-Aware Applications", Proc. of the 13th EUNICE Open European Summer School 2007 (EUNICE 2007), LNCS 4606, July 2007.
- L.O Bonino da Silva Santos, F. Ramparany, P. Dockhorn Costa, P. Vink, R. Etter, T. Broens, "A Service Architecture for Context Awareness and Reaction Provisioning", 2nd Modeling, Design, and Analysis for Service-Oriented Architecture Workshop (MDA4SOA 2007) co-located with the 2007 IEEE Int'l Conference on Services Computing (SCC 2007) and the 2007 IEEE Int'l Conference on Web Services (ICWS 2007), Salt Lake City, USA, July 2007.
- P. Dockhorn Costa, J.P. Andrade Almeida, L. Ferreira Pires, M. van Sinderen, "Situation Specification and Realization in Rule-Based Context-Aware Applications", 7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2007), LNCS 4531, Paphos, Cyprus, 2007.
- P. Dockhorn Costa, G. Guizzardi, J.P. Andrade Almeida, L. Ferreira Pires, M. van Sinderen, "Situations in Conceptual Modeling of Context", workshop on Vocabularies, Ontologies, and Rules for the Enterprise (VORTE 2006) at IEEE EDOC 2006, IEEE Computer Society Press.
- P. Dockhorn Costa, J.P.A. Almeida, L. Ferreira Pires, G. Guizzardi, and M. van Sinderen, "Towards Conceptual Foundations for Context-Aware Applications", Proc. of the Third International Workshop on Modeling and Retrieval of Context (MRC06), Boston, USA, 2006.

- R. Etter, P. Dockhorn Costa, T. Broens, “A Rule-Based Approach Towards Context-Aware User Notification Services”, Proc. of the IEEE International Conference on Pervasive Services 2006, Lyon, France, June 2006.
- P. Dockhorn Costa, L. Ferreira Pires, M. van Sinderen, T. Broens, “Controlling Services in a Mobile Context-Aware Infrastructure”, Proc. of the Second Workshop on Context Awareness for Proactive Systems (CAPS 2006), Kassel, Germany, June 2006, pp. 153-167.
- P. Dockhorn Costa, L. Ferreira Pires, M. van Sinderen, “Architectural Support for Mobile Context-Aware Applications”, book chapter of the Handbook of Research on Mobile Multimedia, Idea Group Inc., 2005.
- S. Pokraev, J. Koolwaaij, M. van Setten, T. Broens, P. Dockhorn Costa, M. Wibbels, P. Ebben, P. Strating, “Service Platform for Rapid Development and Deployment of Context-Aware, Mobile Applications”, Proc. of International Conference on Webservices (ICWS'05), Orlando, Florida, USA, 2005.
- P. Dockhorn Costa, L. Ferreira Pires, M. van Sinderen, “Architectural Patterns for Context-Aware Services Platforms”, Proc. of the Second International Workshop on Ubiquitous Computing (IWUC 2005), Miami, May 2005, pp. 3-19.
- P. Dockhorn Costa, L. Ferreira Pires, and M. van Sinderen, “Designing a Configurable Services Platform for Mobile Context-Aware Applications”, Int’l Journal of Pervasive Computing and Communications (JPCC), 1(1):13-25, Troubador Publishing, 2005.
- T. Broens, S. Pokraev, M. van Sinderen, J. Koolwaaij, P. Dockhorn Costa, “Context-aware, ontology-based, service discovery”, Proc. of the Second European Symposium on Ambient Intelligence (EUSAI 2004), LNCS 3295, Eindhoven, The Netherlands, November 8-10, 2004, pp. 72-83.
- P. Dockhorn Costa, L. Ferreira Pires, M. van Sinderen, and D. Rios, “Services Platforms for Context-Aware Applications”, Proc. 2nd European Symp. on Ambient Intelligence (EUSAI 2004), LNCS 3295, The Netherlands, 2004, pp. 363-366.
- P. Dockhorn Costa, L. Ferreira Pires, M. van Sinderen, and J. Pereira Filho, “Towards a Service Platform for Mobile Context-Aware Applications”, Proc. 1st Intl. Workshop on Ubiquitous Computing (IWUC 2004), Portugal, 2004, pp. 48-61.
- D. Rios, P. Dockhorn Costa, G. Guizzardi, L. Ferreira Pires, J.G. Pereira Filho, M. van Sinderen, “Using Ontologies for Modeling Context-Aware Services Platforms”, Workshop on Ontologies to Complement Software Architectures (OOPSLA 2003), Anaheim, CA, October 26-30, 2003.
- P. Dockhorn Costa, J.G. Pereira Filho, M. van Sinderen, “Architectural Requirements for Building Context-Aware Services Platforms”, Proc. of 9th Open European Summer School and IFIP Workshop on Next Generation Networks (EUNICE 2003), Hungary, September 2003, pp. 62-70.