

A Framework for Context-Aware Pervasive Computing Applications

by

Karen Henricksen
B.InfTech (Honours)

A thesis submitted to

The School of Information Technology
and Electrical Engineering
The University of Queensland

for the degree of

Doctor of Philosophy

September 2003

Declaration

I declare that the work presented in this thesis is, to the best of my knowledge and belief, original, except as acknowledged in the text, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

Karen Henricksen
Brisbane, September 2003

Abstract

The emergence of new types of mobile and embedded computing devices and developments in wireless networking are broadening the domain of computing from the workplace and home office to other facets of everyday life. This trend is expected to lead to a proliferation of pervasive computing environments, in which inexpensive, interconnected computing devices are ubiquitous and capable of supporting users in a range of tasks. It is widely accepted that the success of pervasive computing technologies will require a radical design shift, and that it is not sufficient to simply extrapolate from existing desktop computing technologies. In particular, pervasive computing demands applications that are capable of operating in highly dynamic environments and of placing fewer demands on user attention. In order to meet these requirements, pervasive computing applications need to be sensitive to the context of use, including the location, time and activities of the user.

Currently, the programming of context-aware applications represents a complex and error-prone task, while modification to support changing user requirements or a changing set of context information is usually prohibitively difficult. Consequently, context-aware applications are explored largely in laboratory settings, and remain some distance from widespread acceptance and use. In order to remedy this situation, there is a need for better understanding of the design process associated with context-aware applications, improved programming models that lead to highly flexible and customisable applications, and infrastructural support for tasks such as gathering and management of context information. This thesis presents a framework that addresses these issues. The framework integrates a set of original conceptual foundations, including context and preference modelling techniques, with a software architecture that implements context and preference management functions and provides programming support in the form of a toolkit.

The thesis makes several important research contributions.

First, it presents a novel characterisation of context information in pervasive computing systems, covering (among other features) temporal aspects and various types and sources of uncertainty.

Second, it proposes two complementary approaches to context modelling. The first modelling approach, CML, provides a graphical notation that supports the ex-

ploration and specification of an application's context requirements by the designer. CML represents context information in terms of facts, and has a strong formal basis that enables a straightforward mapping to a context management system built around a relational database. The second approach, termed the situation abstraction, allows contexts to be described in selective, high-level terms as constraints upon a fact-based CML model. Situations are well suited for use in context querying and as programming abstractions.

Third, the thesis presents a pair of programming models that can be used in conjunction with the situation abstraction. The first model, which enables the triggering of actions in response to context changes, has been widely used previously in the development of adaptive and context-aware software, but is reformulated here to accommodate uncertain context information. The second model, which supports choice amongst alternative actions based on the context and preferences of the user (termed branching), is unique to this thesis, and is developed in conjunction with a novel preference modelling approach that allows users to easily express and combine context-dependent requirements.

Fourth, the thesis proposes a software architecture for context-aware systems, which combines toolkit support for the two programming models with software components that perform gathering and processing of context information from a variety of sources, and management of both context and preference information.

Finally, the thesis presents a case study that evaluates a partial implementation of the architecture and its underlying conceptual foundations. This involves the development of a context-aware communication platform that supports choice of communication channels for interactions between users based on the contexts and preferences of the participants. The case study validates the architecture, the context and preference modelling approaches and the branching model, and illustrates the process and issues involved in the design of context-aware software.

Acknowledgements

Many people assisted with the preparation of this thesis, both directly and indirectly. First and foremost, I would like to thank my principal advisor, Associate Professor Jadwiga Indulska, who provided tireless support, encouragement and inspiration throughout my Ph.D. candidature. It was her belief in me, and her persistence and endless good sense, that enabled me to come as far as I have.

My associate advisor, Dr Andry Rakotonirainy, was also an invaluable mentor, and first inspired my interest in context-awareness. I am extremely grateful to him, not only for setting me on this path, but also for his habit of challenging me by asking the difficult questions.

I would like to thank the CRC for Enterprise Distributed Systems Technology (DSTC) for providing me with assistance in many forms, but especially for enabling me to be part of a friendly and supportive research environment during my studies. I am indebted to many DSTC staff and students for their advice and encouragement over the years.

Many people provided feedback on early drafts of this thesis. I would particularly like to acknowledge the thoughtful contributions of Jadwiga Indulska, Andry Rakotonirainy, Matt Henricksen and Ted McFadden.

To my fellow students and friends in GP South, including Ricky, Clinton, Belinda and Ryan, I would like to express my thanks for all the chats, advice and laughter. These things helped to preserve my sanity, and at difficult times gave me a better sense of perspective.

Finally, I would like to thank my family for supporting and believing in me throughout many years of study. I am especially grateful to my parents, Ray and Marion, for always being there for me as a source of inspiration and strength, and to Random, my West Highland Terrier, for his unfailing affection and his special skill in lifting my spirits.

Additional publications relevant to the thesis

1. K. Henriksen and J. Indulska. Adapting the Web interface: An adaptive Web browser. In *2nd Australasian User Interface Conference (AUIC 2001)*, Australian Computer Science Communications, 23(5):21-28. IEEE Computer Society, 2001.
2. K. Henriksen, J. Indulska and A. Rakotonirainy. Infrastructure for pervasive computing: Challenges. In *Informatik 2001 Workshop on Pervasive Computing*, 214-222. Vienna, September 2001.
3. K. Henriksen, J. Indulska and A. Rakotonirainy. Modeling context information in pervasive computing systems. In *1st International Conference on Pervasive Computing (Pervasive)*, Volume 2414 of Lecture Notes in Computer Science, 167-180. Springer, 2002.
4. J. Indulska, R. Robinson, A. Rakotonirainy and K. Henriksen. Experiences in using CC/PP in context-aware systems. In *4th International Conference on Mobile Data Management (MDM)*, Volume 2574 of Lecture Notes in Computer Science, 247-261. Springer, 2003.
5. K. Henriksen, J. Indulska and A. Rakotonirainy. Generating context management infrastructure from context models. In *4th International Conference on Mobile Data Management (MDM) - Industrial Track*. Melbourne, January 2003.
6. J. Indulska, T. McFadden, M. Kind and K. Henriksen. Scalable location management for context-aware systems. To appear in *4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, Paris, November 2003.
7. K. Henriksen and J. Indulska. A software engineering framework for context-aware pervasive computing. To appear in *2nd IEEE Conference on Pervasive Computing and Communications (PerCom)*, Orlando, March 2004.

8. K. Henriksen and J. Indulska. Modelling and Using Imperfect Context Information. To appear in *PerCom 2004 Workshop on Context Modeling and Reasoning (CoMoRea)*, Orlando, March 2004.

Contents

1	Introduction	1
1.1	Motivation	1
1.1.1	Autonomy	2
1.1.2	Dynamic computing environments	2
1.1.3	Dynamic user requirements	2
1.1.4	Scalability	2
1.1.5	Resource limitations	2
1.2	The role of context-awareness	3
1.3	Thesis statement	4
1.4	Approach	5
1.4.1	Context modelling approaches	5
1.4.2	Programming models	7
1.4.3	Preference model	7
1.5	Applications	7
1.5.1	Context-aware communication	8
1.5.2	Context-aware tour guides	9
1.5.3	Context-aware reminders	9
1.5.4	Intelligent environments	10
1.6	Structure of the thesis	10
2	Defining context	13
2.1	Defining context in the scope of pervasive computing	13
2.2	Characteristics of context information	15
2.2.1	Heterogeneity	15
2.2.2	Uncertainty	16
2.2.3	Histories	17
2.2.4	Dependencies	18
2.3	Alternative definitions of context	19

3	Related work	21
3.1	Context gathering infrastructure	22
3.1.1	Overview	22
3.1.2	Discussion	24
3.2	Context models and servers	24
3.2.1	Overview	24
3.2.2	Discussion	27
3.3	Programming abstractions for context-aware computing	27
3.3.1	Overview	27
3.3.2	Discussion	30
3.4	Design tools for context-aware software	31
3.4.1	Overview	31
3.4.2	Discussion	32
3.5	Preference and user modelling	32
3.5.1	Overview	32
3.5.2	Discussion	36
3.6	Summary and conclusions	36
4	Context modelling	39
4.1	Terminology	39
4.2	Motivation and approach	41
4.3	Introduction to ORM	43
4.3.1	Modelling concepts	43
4.3.2	Modelling methodology	45
4.4	Context modelling concepts	47
4.4.1	Classification of fact types	47
4.4.2	Temporal fact types	50
4.4.3	Quality	52
4.4.4	Alternatives	56
4.4.5	Fact dependencies	58
4.4.6	Summary	59
4.5	Context modelling process	60
4.6	Relational mapping	62
4.6.1	Overview of the relational model	62
4.6.2	Mapping approach	64
4.6.3	Classification of fact types	65
4.6.4	Temporal fact types	67
4.6.5	Quality	69
4.6.6	Alternatives	70
4.6.7	Fact dependencies	71

4.6.8	Summary	74
4.6.9	Interpretation	76
4.7	Discussion and future work	78
5	Programming abstractions	81
5.1	Motivation and approach	82
5.2	The situation abstraction	83
5.2.1	Overview	83
5.2.2	Defining situations	84
5.2.3	Examples	86
5.2.4	Combining situations	89
5.2.5	Handling uncertainty	91
5.2.6	Efficiency	93
5.3	Programming models	95
5.3.1	Triggering model	95
5.3.2	Branching model	100
5.4	The preference abstraction	103
5.4.1	Overview and requirements	103
5.4.2	Preference model	105
5.4.3	Combining preferences	108
5.4.4	Programming with preferences	112
5.4.5	Learning preferences	117
5.5	Summary of contributions	118
5.6	Discussion and future work	119
6	An architecture for context-aware pervasive systems	121
6.1	An architecture for context-aware systems	121
6.1.1	Context gathering layer	122
6.1.2	Context reception layer	124
6.1.3	Context management layer	125
6.1.4	Query layer	126
6.1.5	Adaptation layer	127
6.1.6	Application layer	129
6.1.7	Summary	130
6.2	Discussion	130
6.2.1	Autonomy	133
6.2.2	Dynamic computing environments	133
6.2.3	Dynamic user requirements	134
6.2.4	Scalability	135
6.2.5	Resource limitations	136

6.3	An implementation of the architecture	137
6.3.1	Organisation	137
6.3.2	Implementation of the context management layer	139
6.3.3	Implementation of the query layer	140
6.3.4	Implementation of the adaptation layer	141
6.3.5	Implementation of the application layer	141
6.4	Discussion and future work	142
7	Case study: Context-aware communication	145
7.1	Introduction	146
7.2	Objectives	146
7.3	Requirements analysis	147
7.4	Design considerations	148
7.4.1	Autonomy and control	148
7.4.2	Privacy	149
7.5	Design process	150
7.6	Core functionality	150
7.6.1	Overview	152
7.6.2	Scope	152
7.7	Context-aware behaviour	153
7.8	Context modelling	155
7.9	Preference modelling	158
7.10	High-level design	162
7.11	Implementation	163
7.12	Discussion	164
7.13	Conclusions and future work	167
8	Discussion and conclusions	169
8.1	Summary of contributions	169
8.2	Future work	172
8.2.1	Context management	172
8.2.2	Design tools and methodologies	172
8.2.3	Evaluation	173
A	Syntax and semantics of the situation abstraction	175
A.1	Syntax	175
A.2	Semantics	176
A.3	Composite situations	179
A.4	The extended situation abstraction	180

List of Figures

1.1	Components of the conceptual framework and infrastructure	6
4.1	Context modelling approach used in this thesis.	40
4.2	FORM modelling example.	44
4.3	Context classification examples.	49
4.4	Temporal fact type example.	52
4.5	Annotation of a fact type with quality indicators.	55
4.6	Alternative fact type example.	57
4.7	Fact dependency example.	59
4.8	Summary of notation.	60
4.9	Relational mapping example.	66
4.10	Mapping a derived fact type to the relational model.	68
4.11	Mapping a temporal fact type to the relational model.	69
4.12	Mapping quality annotations to the relational model.	70
4.13	Mapping an alternative fact type to the relational model.	71
4.14	Mapping a dependency to the relational model.	73
5.1	A context model for the context-aware communication scenario.	87
5.2	Example situation predicates.	88
5.3	State transitions for a situation S	98
5.4	Example triggers.	101
5.5	Example preferences.	107
5.6	Example preference sets.	109
5.7	Example composite preferences.	113
5.8	A preference hierarchy.	114
5.9	Selected methods of a programming toolkit that implements the SBB model.	115
6.1	Architecture.	123
6.2	Example queries.	128
6.3	Organisation of the prototype.	138

7.1	The main window of the Mercury user interface.	151
7.2	New query dialog.	154
7.3	New interaction dialog.	154
7.4	Channel selection dialog.	154
7.5	Channel selection dialog, showing a drop-down list of options.	154
7.6	Channel negotiation protocol.	155
7.7	The detailed context model.	157
7.8	Mercury's core set of situation predicates.	159
7.9	Mercury's default initiator and recipient preferences.	160
7.10	Mercury's generic and composite preferences.	161
7.11	High level design of the prototype.	163

List of Tables

2.1	Typical properties of context information.	16
4.1	Example instantiation of the <i>has process number</i> fact type.	45
4.2	Example instantiation of the <i>executes on</i> fact type.	45
4.3	Example instantiation of the <i>has user</i> fact type.	45
4.4	Example instantiation of the <i>used by...from...to</i> fact type.	45
4.5	Common quality parameters.	54
4.6	Example instantiation of the fact type shown in Figure 4.5.	55
4.7	Example instantiation of the fact type shown in Figure 4.6.	57
4.8	Example instantiation of the <i>EngagedIn</i> relation.	74
4.9	Example instantiation of the <i>LocatedAt</i> relation.	74
4.10	Example instantiation of the <i>LocatedAtDependents</i> relation.	74
5.1	Truth table for the three-valued logic.	93
6.1	Components of the programming toolkit.	131
6.2	Functions of the six architectural layers.	132
7.1	Relative sizes of the implementation components.	164

Chapter 1

Introduction

1.1 Motivation

Current trends in mobile computing devices and wireless networking technologies are helping to broaden the domain of computing. Increasingly, computing is occurring within settings far removed from the desktop, such as in the car or the shopping centre, where computing applications compete with other tasks for user attention. The spread of computing to these new environments, supported by a variety of interconnected mobile and embedded devices, is commonly referred to as pervasive or ubiquitous computing.

Today, computing is increasingly being carried out using small devices, such as PDAs and mobile telephones, which support mobile data entry applications, email and text messaging, Web browsing and telephony. Embedded devices are also increasingly being used to enable novel applications such as automated inventory management, which involves the augmentation of goods with chips that can be interrogated by remote readers and programmed with information such as shipping and expiry dates. Similarly, computational capabilities are being added to a variety of consumer products, leading to appliances such as Internet-enabled refrigerators that support browsing of recipes and online grocery shopping, and air-conditioners that can be controlled remotely through Web or telephone interfaces.

In the future, the applications of computing technology will continue to multiply, with the result that humans will inhabit environments in which vast numbers of invisible computers embedded in the surroundings, together with more traditional computing devices, will provide seamless support for a broad spectrum of human activities. As the numbers of computers and applications grow, a variety of new challenges will be faced.

1.1.1 Autonomy

The increasing ratio of computing applications to humans is steadily reducing the amount of attention that users can expend upon each application. This trend coincides with the shift of computing away from the desktop and into environments in which human-computer interaction must take a back seat to other activities. These factors have led to the proposal of “invisible computing” [1], a novel design approach that sees computing blended seamlessly into physical environments such that computing applications provide users with unobtrusive support for the tasks at hand. The design of applications that possess the required degrees of autonomy and invisibility without removing the user’s sense of control presents a significant challenge.

1.1.2 Dynamic computing environments

The mobility of computing devices, applications and people leads to highly dynamic computing environments. Unlike desktop applications, which rely on a carefully configured and largely static set of resources, pervasive computing applications are subjected to changes in available resources such as network connectivity and input and output devices. Moreover, they are frequently required to cooperate spontaneously and opportunistically with previously unknown software services in order to accomplish tasks on behalf of users. Thus, pervasive computing software must be highly adaptive and flexible.

1.1.3 Dynamic user requirements

Similarly, pervasive computing applications must accommodate changes in user requirements that arise as the user’s activities and goals evolve. As an example, an application may need to modify its style of output following a transition from an office environment to a moving vehicle, in order to be less intrusive.

1.1.4 Scalability

As computing devices and applications continue to proliferate, substantial scalability problems arise. In response, the designers of pervasive computing software must overcome these problems by paying special care to the appropriate distribution of functionality, the scalability of communication paradigms, the application of information reduction and filtering techniques, and so on.

1.1.5 Resource limitations

Pervasive computing environments are characterised by the presence of heterogeneous computing devices, ranging from resource-rich to resource-poor. Conse-

quently, the success of software systems in these environments is dependent on their ability to adapt to, and overcome, resource limitations. Current generation mobile and embedded devices frequently suffer from severe restrictions with regard to communication capabilities and bandwidth, battery lifetime, memory capacities, processing power, and input and output capabilities. Advances in computing resources (processing speed, memory, etc.) are far outstripping similar advances in battery technology. Therefore, power consumption is likely to become an especially prominent design consideration in the future. Resources that have significant energy demands, such as wireless communications, will similarly be constrained. Passive devices, such as RFID tags, that can be powered remotely using induction are likely to play a significant role in future computing environments.

Input and output devices are also of key importance. Traditional I/O mechanisms (keyboards, pointing devices and desktop displays) will have reduced importance in pervasive computing environments, while voice-based input and output, heads-up displays, and so on, are likely to play more prominent roles. Opportunistic use of the devices present in the surrounding environment (for example, a wall mounted display in a meeting room) will also become increasingly useful.

1.2 The role of context-awareness

The set of challenges presented by pervasive computing necessitates a radically new application design approach. The key requirements of pervasive computing applications are flexibility and autonomy, while scalability and resource limitations represent important design constraints. *Context-awareness* has been proposed as a novel design approach that satisfies the flexibility and autonomy requirements. It involves the exploitation of context information by applications in order to reduce reliance on user input and promote adaptation to dynamic factors in computing environments and user requirements. Context information describes relevant aspects of the surrounding physical and computing environments, such as the location and activity of the user, the presence of other people nearby, the time of day, and the set of available computing resources. This information can be derived from a variety of sources, such as hardware and software sensors, or user profiles. Much of the recent research in context-aware computing has focused exclusively on sensor-derived context information, such as location information obtained from wireless positioning devices [2–6].

While pervasive computing has become a hot research topic in recent years, context-awareness remains in its infancy. A variety of prototypical context-aware applications, such as context-aware tour guides, reminder applications and communication tools, have recently appeared [7–20]. However, these represent fairly simple uses of narrow sets of context information. Moreover, while a few of the applications

have been evaluated in (somewhat) realistic usage scenarios, there are currently no commercially successful examples of context-awareness. This situation is largely due to the inherent software engineering challenges associated with:

- the analysis and specification of application requirements for context information;
- the acquisition of the required information from suitable sources, such as sensors, and subsequent management and dissemination of relevant information to applications; and
- the design and implementation of suitable context-aware application behaviours that meet the unique usability requirements of pervasive computing environments.

The bulk of recent research in context-awareness has focused on the development of infrastructure to overcome the second problem. Particular emphasis has been placed on the construction of context-gathering infrastructures that obtain and abstract context information from sensors [21–23], and context servers that manage repositories of context information that can be easily queried by applications [2, 24–27].

In contrast, the first and third issues have received little attention, with generic software analysis, design and programming methodologies generally being applied in the absence of specialised abstractions and models for context-awareness.

1.3 Thesis statement

This thesis contends that traditional methods of software development are ill-equipped to meet the challenges inherent in developing context-aware software for pervasive computing environments. It proposes that the use of specialised tools and abstractions, in conjunction with infrastructural support for tasks such as context gathering, management and dissemination, greatly simplifies the construction of context-aware applications.

The contribution of this thesis lies in the development of a conceptual framework, and a corresponding software infrastructure, that supports analysis, design and implementation tasks associated with context-awareness. At the core of this framework are novel context modelling approaches that allow context to be described at varying levels of abstraction and programming models that can be used in conjunction with these.

1.4 Approach

The research presented in this thesis has both theoretical and practical components. The former takes the form of a conceptual framework consisting of:

- context modelling approaches that enable the description of contexts in terms of fine-grained facts and more abstract situations;
- programming models, founded on the situation abstraction, that support triggering in response to context changes and context-dependent branching; and
- a model of user preferences that can be used in conjunction with the branching model to support flexibility and personalisation.

The components of the conceptual framework are mapped to a software infrastructure that assists with both implementation and run-time tasks, as shown in Figure 1.1. The infrastructure offers support for the two programming models in the form of a programming toolkit that enables developers to insert triggers and branches into program code. Similarly, the preference and context modelling approaches are mapped to management systems that gather, store and respond to queries about preference and context information, respectively. The situation abstraction is mapped to a situation-based query interface that can be used to interrogate the context repository maintained by the context manager in high-level terms.

The following sections describe the components of the conceptual framework in further detail.

1.4.1 Context modelling approaches

A two-tiered context modelling approach forms the heart of the framework. At a detailed level, context is expressed in terms of atomic facts, and, at a more abstract level, in terms of situations. The two modelling approaches serve complementary purposes.

The fact-based modelling approach functions as a useful tool for the analysis task involving the identification and specification of an application's context requirements. This task is accomplished using a graphical modelling notation to produce a context model in terms of abstract fact types. The model is instantiated at run time by a set of facts maintained by the context management component shown in Figure 1.1. These facts capture the context at a level of abstraction that roughly matches that employed by the producers of the context information, such as sensors and user profiles (although some processing or interpretation may be required to transform sensor - and other - inputs into facts).

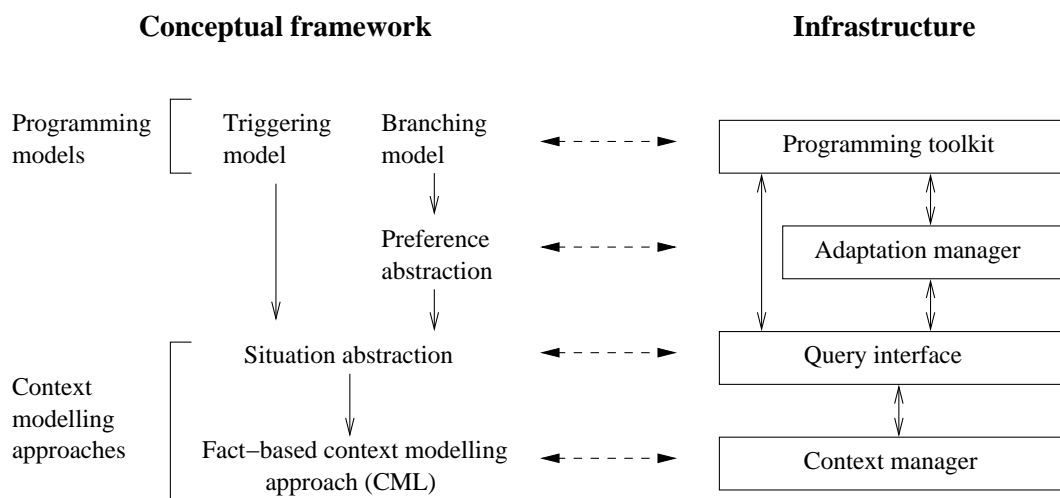


Figure 1.1: Components of the conceptual framework (left) and their mapping to infrastructural components (right). Relationships between the components are shown by arrows. The situation abstraction is based on the fact-based modelling approach, and in turn forms one of the components of the preference model. The trigger-based programming model builds directly upon the situation abstraction, while the branching model builds on the preference model. The context modelling approaches and the preference abstraction receive implementation support in the form of context and adaptation managers, respectively; the situation abstraction is supported by a query interface to the context manager; and the programming models are mapped to a programming toolkit.

The situation-based model enables context to be described in terms of scenarios or conditions that match the level of abstraction required by the application programmer. Consequently, this model forms an appropriate basis for expressing high-level requests to the context management system, in the form of queries and event subscriptions. Situations are specified as predicates over zero or more facts, and can be easily combined to represent increasingly complex scenarios.

1.4.2 Programming models

Two complementary programming models can be used in conjunction with the situation abstraction. The trigger-based model enables a situation to be associated with a set of actions that are automatically executed upon a state change, such as the entering or exiting of the situation. The branching model enables choices to be inserted at relevant points in the code (in the manner of an extremely flexible form of case statement), such that the choices - termed branches - that are selected at any given time are determined by the context.

As trigger-based models have been widely studied previously [23, 28–35], this thesis places greater emphasis on the branching model.

1.4.3 Preference model

As mentioned previously, one of the primary challenges in developing context-aware applications lies in achieving an appropriate balance between application autonomy and user control. In light of this challenge, the branching-based programming model developed in this thesis employs user preferences, in conjunction with context information, as a basis for determining the branch(es) that are selected and executed. The preference model allows users to express the relative desirability of alternative actions in relation to the context, and also enables users to forbid or force choices in certain contexts. Preferences are expressed in a format that allows users to easily specify (and later modify) fine-grained preferences, which can be combined to support complex and flexible application behaviour.

1.5 Applications

The conceptual framework and infrastructure presented in this thesis are designed to be applicable to *all* pervasive computing applications that can usefully benefit from context-awareness. However, there are currently significant barriers to evaluating the extent to which this goal is met:

- As context-awareness remains in its infancy and has yet to achieve widespread acceptance, the really compelling uses of context are not presently known [36].

- Truly pervasive computing has not yet arrived. Therefore, there is great difficulty associated with developing and evaluating prototypical context-aware applications within realistic usage scenarios and appropriate hardware and software environments [37].

In the face of these challenges, the target applications of this research are those context-aware applications that have been widely studied in the past few years: context-aware communication tools, tour guides, reminder applications and intelligent environments¹. These are outlined in the following sections. Context-aware communication is arguably one of the most compelling and widely studied applications [39, 40], and is adopted as a running example throughout this thesis.

1.5.1 Context-aware communication

With the increasing ubiquity of mobile telephones, their intrusiveness is becoming evident. To circumvent this problem, people resort to switching their devices off altogether in environments such as meetings and theatres, or to using silent modes that provide unobtrusive alerts of incoming calls which can be ignored if desired. However, these solutions have at least two shortcomings. First, missed calls are common, as the caller has no way of knowing before dialling whether the other party is available and willing to talk. Second, the recipient of the call typically has very limited information upon which to judge the importance of incoming calls². Context-awareness provides both the caller and the callee (or their software agents) with improved knowledge of the circumstances surrounding the call and the activities of the other participant, and presents a compelling solution to these problems³.

Telephony has been recognised as an important application of context awareness for over a decade [41], and has been widely researched [7, 23, 42, 43]; however, the potential uses of context information in the communication domain are much more extensive [44]. Other domains that have been explored in recent years include text-based messaging and chat programs [8–10, 45, 46], home intercoms [11, 39] and multimedia communication between staff in a hospital setting [47]. The framework presented in this thesis is designed to be sufficiently generic to support all of these application domains. However, the communication application developed as a case study in this thesis is broader, and involves an integrated communication platform that incorporates a range of traditional interaction protocols such as email, telephony, text messaging and videoconferencing. Within this platform, the context

¹This list of context-aware applications is by no means exhaustive, but includes the most widely researched applications. Further examples can be found in a survey by Chen and Kotz [38].

²Usually the callee knows only the telephone number from which the call originates, but even this information is unavailable when the call is from an unlisted number.

³Clearly there are privacy concerns associated with exposing very detailed information information about user activities, but these can be overcome using an appropriate application model, as discussed in Chapter 7.

and the preferences of users form the basis for the choice of protocol and communication device for each interaction, as described in the following scenario:

Mark has finished reviewing a paper for Emma, and wishes to share his comments with her. He instructs his communication agent to initiate a discussion with Emma. Emma is in a meeting with a student, so her agent determines on her behalf that she should not be interrupted. The agent recommends that Mark contact Emma by email. Mark composes an email on the workstation he is currently using, and his agent dispatches it according to the instructions of Emma's agent to her work email address.

A few minutes later, Emma's supervisor, Michelle, wants to know whether the report she has requested is ready. Emma's agent has been instructed that calls from Michelle have high priority, and decides that the query should be answered immediately. The agent suggests that Michelle telephone Emma on her office number. Michelle's agent establishes the call using the mobile phone that Michelle is carrying with her.

An implementation of a communication application that supports this and broader scenarios (involving multi-party communication, a wide choice of communication channels, and so on) is presented in Chapter 7.

1.5.2 Context-aware tour guides

Context-awareness also has compelling uses within the domain of information retrieval. Context-dependent information retrieval has been studied both in a general sense [48,49], and as a basis for the construction of a variety of context-aware guides. Two of the best known guide applications are GUIDE [12] and Cyberguide [13], which present tourists with location-dependent information and services on hand-held devices. GUIDE provides visitors to the city of Lancaster with information about nearby attractions, tailored city tours, interactive services such as hotel booking facilities, and a messaging service, all within a Web-based navigational interface. Cyberguide explores the provision of similar services (maps, location-dependent information, communication tools and positioning information) to visitors within single building and campus-wide settings. A variety of context-aware guides have also been constructed for museum, exhibition and conference environments [14,15,50–53].

1.5.3 Context-aware reminders

A second class of information retrieval application that has been widely researched is concerned with producing context-triggered reminders. The applications belonging to this class allow users to write reminder notes to themselves or other users, such that each note is attached to a given context and presented to the intended recipient

when this context is detected. CybreMinder [16] employs reminder messages resembling emails: these have a recipient, subject, priority, expiration time and message body, and are attached to a *situation*, specified in terms of an arbitrarily complex set of conditions that define the delivery context. ComMotion [17] employs a narrower model of context, based on location and time, and delivers reminder messages using speech synthesis. Reminders can be created as entries in to-do lists, or as email messages. Finally, MemoClip [54] is a purpose-build appliance that provides location-triggered reminders. Resembling a paging device, the MemoClip beeps to alert users of reminder messages, displays short text messages on a small screen and tracks location using a simple infrared positioning system.

1.5.4 Intelligent environments

Perhaps the most ambitious use of context information to date is demonstrated by a variety of projects that are concerned with the embedding of computing and context sensing infrastructure into environments such as homes, classrooms and meeting rooms. The goals of these projects are diverse. Several aim to monitor the activities of the elderly within their homes or assisted living settings, with the objective of allowing them to retain their independence, while ensuring that emergencies are quickly detected [55–61]. Others use context information to automate the control of environmental conditions, including ambient lighting and temperature, and of the various appliances present in the environment [62–64]. Finally, the projects involving classrooms and meeting rooms are principally concerned with facilitating remote collaboration [65] or with providing integrated capture and note-taking facilities, enabling rich records to be retained of the proceedings that occur within the specially instrumented environments [66–68].

1.6 Structure of the thesis

The thesis is organised as follows.

Chapter 2 formalises the assumptions of the research presented in this thesis with regard to the nature of context and context information, and positions these assumptions in relation to other research concerned with context.

Chapter 3 surveys relevant literature in context-awareness and related research areas, with particular focus on context acquisition, management, modelling and dissemination, programming abstractions, design tools, and preference and user modelling techniques.

Chapters 4 and 5 develop a novel theoretical framework for context-aware computing. At the heart of this framework is a context modelling approach which supports the description of contexts in terms of fine-grained facts; this is the subject

of Chapter 4. A graphical modelling notation and design methodology, created to support developers in the task of exploring and formally specifying the context requirements of an application, are presented. The formal basis for the model is also developed; this supports reasoning about contexts and enables a straightforward mapping of a context model to a relational database schema.

In Chapter 5, a situation-based model of context is constructed upon the formal foundation of the fact-based approach. This model enables contexts to be described at a level of abstraction that is natural to both application programmers and users. A pair of complementary programming models that can be used in conjunction with the situation abstraction, based on triggering and branching, are outlined. The branching model is explored in detail, and a novel model of user preferences that supports highly dynamic branching behaviour, tailored to the needs of individual users, is presented.

Chapter 6 develops a system architecture for context-aware applications that integrates the modelling and programming approaches developed in Chapters 4 and 5. The design of this architecture is inspired by the five challenges introduced in Section 1.1, and is validated by both the implementation that is presented at the conclusion of the chapter and the case study introduced in Chapter 7.

The case study develops the context-aware communication application described in Section 1.5 to demonstrate, in a concrete manner, the variety of software engineering issues involved in the development of context-aware software, and the combined utility of the modelling concepts, programming techniques and software infrastructure developed in Chapters 4, 5 and 6 in addressing these.

Finally, Chapter 8 summarises the contributions of the thesis and outlines topics for future research.

Chapter 2

Defining context

The term *context* is loaded with a variety of different meanings. The goals of this chapter are to characterise some of the most common of these, to clearly define the interpretation that is used within the scope of this thesis, and to outline some of the pertinent features of context information in pervasive computing environments which have an impact on the design of context-aware systems and applications.

The structure of this chapter is as follows. Section 2.1 characterises common definitions assigned to context by the pervasive computing community; these are closely aligned with the interpretation that is adopted within this thesis. Next, Section 2.2 outlines some of the key features and constraints of context information in pervasive systems that arise as a result of the means by which the information is gathered, disseminated and used. These serve as key considerations in the design of the abstractions and infrastructure presented in later chapters. Finally, Section 2.3 surveys some alternative views of context, from research areas such as artificial intelligence and information retrieval, and contrasts these with the pervasive computing perspective.

2.1 Defining context in the scope of pervasive computing

As described in Section 1.2, *context-awareness* emerged from the field of pervasive or ubiquitous computing as a technique for imbuing applications with an awareness of their surroundings, in order to achieve a high degree of autonomy and flexibility. The chief goal is to reduce the burden on users to interact frequently with and to drive their applications, by creating software that is unintrusive and largely invisible. This design goal is often termed *invisible computing* [1], and is becoming more relevant as computing spreads to environments in which user attention is at a premium (for example, within motor vehicles), and as the ratio of applications to users grows.

Despite the appearance of a spate of context-aware applications in recent years,

the concept of *context* remains ill-defined. Dey [69] presents a survey of alternative views of context, which are largely imprecise and indirect, typically defining context by synonym or example. Context has been variously characterised as an application’s environment or situation [25, 70], and as a combination of features of the execution environment, including computing, user and physical features [33]. Dey also offers the following definition, which is perhaps now the most widely accepted:

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

The meaning of “the situation of an entity” is never precisely defined by Dey, but instead is illustrated through simple examples.

One of the reasons for the lack of consensus and precision in the above definitions, including that of Dey, is the lack of clear separation between the concepts of *context*, *context modelling* and *context information*. This distinction is a very important one. While context represents a nebulous concept that is difficult to define or bound, context models and information are necessarily well defined and understood. Fortunately, when constructing context-aware systems, it is the latter two concepts that are primarily of interest.

Within this thesis, context information is explored as a enabling mechanism for applications to perform tasks on behalf of users in an autonomous and flexible manner. Therefore, context is considered in relation to tasks, rather than with respect to interactions between users and applications, as in the definition of Dey. Specifically, the following interpretations are adopted:

- The *context* of a task is the set of circumstances surrounding it that are potentially of relevance to its completion.
- A *context model* identifies a concrete subset of the context that is realistically attainable from sensors, applications and users and able to be exploited in the execution of the task. The context model that is employed by a given context-aware application is usually explicitly specified by the application developer, but may evolve over time.
- *Context information* is a set of data, gathered from sensors and users, that conforms to a context model. This provides a snapshot that approximates the state, at a given point in time, of the subset of the context encompassed by the model.

The approximate and limited nature of context information should be a crucial consideration when constructing context-aware software. Surprisingly, this has not

been the case in the past, as demonstrated in the survey that appears in the following chapter. Consequently, usability problems arising from unmet expectations about the quality of the available context information are often observed in context-aware applications [17, 71, 72]. It is one of the goals of the research presented in this thesis to address this problem by highlighting some of the constraints associated with context information, and designing context modelling solutions that recognise these. To this end, a discussion of key characteristics of context information appears in the following section.

2.2 Characteristics of context information

2.2.1 Heterogeneity

While many context-aware applications exploit only sensed context information (and some only location data), the most useful applications are usually those that combine sensed with non-sensed information, merging these to form a rich context description. The non-sensed information is frequently obtained from users, often indirectly from user profiles or applications such as scheduling tools that record user activities. Other context information is obtained through derivation mechanisms, such as synthesis of multiple sources of context data or interpretation of a single source to obtain a higher level of abstraction.

The integration of context information from such diverse sources naturally leads to extreme heterogeneity in terms of characteristics such as quality and persistence. The remainder of this section outlines some of the typical properties of sensed, user-supplied and derived context information, while the following section elaborates further on quality issues.

Sensed context information is often highly dynamic, and, even when not changing, is usually updated frequently in response to continuous or periodic sensor output. It is prone to inaccuracies as a result of sensing errors, and at times may be completely unknown, owing to sensor failures, network disconnections or limitations inherent within the sensing technology¹. In addition, when the context is changing rapidly, the delays introduced by distribution and the interpretation process that transforms the sensor output into high-level context information can lead to staleness.

User-supplied context information can be partitioned into static and dynamic information. Intuitively, static information describes persistent properties such as the type of a computing device or communication channel. A high degree of confidence can be assigned to this type of information. In contrast, dynamic context information, which is referred to in this thesis as *profiled* information, can suffer from

¹A common example of this is the inability of GPS receivers to function effectively indoors.

Table 2.1: Typical properties of context information.

<i>Class of information</i>	<i>Persistence</i>	<i>Quality issues</i>	<i>Sources of inaccuracy</i>
Sensed	Low	May be inaccurate, unknown or stale	Sensing errors; sensor failures or network disconnections; delays introduced by distribution and the interpretation process
Static	Forever	Usually none	Human error
Profiled	Moderate	Prone to staleness	Omission of user to update in response to changes
Derived	Variable	Subject to errors and inaccuracies	Imperfect inputs; use of a crude or oversimplified derivation mechanism

staleness as a result of neglect on the part of the user to update the information as it changes. Unless obtained using indirect means (e.g., from scheduling and other applications), it is usually unreasonable to capture properties that change on a daily or even weekly basis as profiled information, because of the burden this places on the user. Profiled context is best used to characterise more enduring properties, such as associations between users and communication channels, or relationships between people.

The characteristics of derived context information are largely determined by the properties of the input data and of the derivation mechanism. Derived information usually retains (and sometimes even magnifies) any inaccuracies present within the input data. Additionally, the use of brittle heuristics or the reliance on crude sensor inputs to infer high-level context information naturally lead to further potential for error.

The typical properties of the four types of context information are summarised in Table 2.1.

2.2.2 Uncertainty

Various sources of inaccuracy and error were characterised in the previous section. Here, the problem of imperfect context information is addressed more formally. It can be partitioned into at least four distinct subproblems. An attribute of the environment covered by the context model (such as the location of a given person at a specified time) is:

- *unknown* when no information about that aspect is available;
- *ambiguous* when several different reports about the attribute exist (for example, when two distinct readings are supplied by separate positioning devices;

these readings might conflict, correspond to overlapping regions, or simply describe the same location at different levels of granularity);

- *imprecise* when the reported state is a correct yet inexact approximation of the true state (for example, when the location of an object is known to be within a given region, but the precise coordinates of the object within this region cannot be pinpointed); or
- *erroneous* when there is a mismatch between the actual state of the environment and the reported one (as for example, when a user becomes separated from his/her positioning device).

It will not always be possible to overcome these four classes of imperfect context information (or even to detect instances of the fourth class); however, context-aware systems can minimise the impact of imperfect information on applications by using appropriate context modelling and management techniques. The survey presented in the following chapter demonstrates that only a small subset of the previous research in context-awareness addresses even *one* of the classes, and, moreover, that none of the surveyed context modelling approaches addresses all four.

2.2.3 Histories

Section 2.2.1 introduced an important distinction between static (user-supplied) context information, and dynamic (sensed, profiled or derived) information. When dealing with the latter, applications may not be solely interested in the current state, but also in future or past states, or changes in state over time. In this thesis, the record of an attribute belonging to a context model over time is termed a history (regardless of whether it covers past states, future states or both).

The uses of historical context information include the following:

- *Prediction.* This is based on extrapolation from past behaviour (allowing, for example, a user's location five minutes hence to be guessed from previous movements). Some useful applications of historical information in adapting fidelity (for example, quality of image rendering in graphics applications), according to resource availability, are described by Narayanan et al. [73]. Salber and Abowd [74] describe further uses, such as the identification of common interests between users based on Web browsing history.
- *Detection of changes.* The invocation of adaptation actions in response to context changes (called triggering) is a widely exploited technique in the programming of context-aware software, as described in Section 1.4.2. Historical context information is required in order to detect many context changes (for example, a temperature change of more than three degrees Celsius within the

past thirty minutes, or the movement of a user from home to the office via arbitrary intermediate destinations).

- *Exploitation of planned context.* When available, information about users' future plans can serve as a useful type of context information. For example, the activities of a user in the next half hour, as indicated by the user's scheduling software, can assist a communication agent in determining whether it is appropriate to admit an incoming telephone call.

In order to support these types of behaviour, techniques for modelling and querying historical context information are required.

2.2.4 Dependencies

A context model typically combines information about diverse yet interrelated entities, such as users, computing devices and the physical environments in which these reside. The model should capture relevant relationships between these (such as proximity of users to their computing devices), as well as relationships that exist among different elements of the context description itself (or more precisely, between their states)². The second type of relationship can be regarded as a type of context metadata, and is referred to as a *context dependency* within this thesis.

A context dependency can be modelled in terms of a binary relation that contains the pair (i_2, i_1) if the state of element i_2 is at least partially determined by the state of element i_1 . Dependencies are induced either as a result of physical laws (as in the case of the dependency of a mobile device's battery lifetime on the bandwidth usage) or as an artifact of the means by which the context information is derived (as in the case of the dependency of user activity information on user location, when the former is computed from a combination of the latter and other relevant cues, such as level of lighting, presence of other users and noise level³).

The importance of recognising context dependencies and exploiting these when performing adaptation is illustrated by Efstratiou et al. [75]. They illustrate a scenario in which uncoordinated adaptation of applications, which ignores the relationship between power consumption and bandwidth, leads to conflicting, and potentially cascading, adaptation actions. In this scenario, an application first detects the need to reduce power usage by decreasing its reliance on bandwidth. Following this action, the application observes a large amount of spare bandwidth, and takes actions to increase its usage. This can again lead to the first action, causing the application to remain indefinitely in a state of instability.

²In the case of our fact-based context modelling approach, these would be relationships between facts.

³Of course, there is a real-world correlation between a person's activity and location, but it is not as simple as this dependency implies, nor as obvious and quantifiable as the relationship between a mobile device's bandwidth and battery lifetime.

Information about dependencies can also be exploited for context management purposes (for example, to proactively trigger updates of possibly stale context information). A discussion of this issue is deferred until Section 4.4.5.

2.3 Alternative definitions of context

Having examined the definitions assigned by the pervasive computing community (and this thesis) to the term context, this section seeks to briefly contrast these with some of the alternative perspectives adopted by other branches of computer science research. The principal goals are to illustrate the clear separation that exists between these various investigations of context, and, in doing so, to better position the research contributions of this thesis. As the applications of context to computer science are extremely broad, this discussion is intended to be illustrative rather than exhaustive.

The concept of context is applied extensively within artificial intelligence. In machine learning, context is used to improve the effectiveness of learning algorithms. For example, Widmer and Kubat [76] describe an approach in which “forgetting” of learned concepts occurs over time to accommodate context change, and previously learned concepts are stored and later retrieved when the context reverts to the earlier state. Turney [77] presents a good review of a broader set of strategies for handling context-sensitivity in machine learning. This demonstrates that the types of context considered in learning tasks (termed contextual features) are highly domain-dependent. These include lighting conditions in the case of image classification, and the speaker’s accent and identity in the case of speech recognition.

In the related domain of knowledge acquisition, the explicit representation of context in knowledge bases is viewed as crucial to the success of expert systems [78]. Context is variously encoded within knowledge bases as premises associated with rules [79], as metadata taking the form of context ontologies [80], or as contextual schemas that integrate context descriptions with prescriptive knowledge that describes the appropriate behaviour for the corresponding context [81]. A relatively formal approach to context modelling is proposed by Lenat, who suggests that context can be described as “a region in some n -dimensional space” [82]. Lenat identifies twelve dimensions that he argues are the most relevant when characterising the applicability of general types of knowledge; these include time, place, culture, topic, granularity and justification.

A related area of research examines the application of context to logical reasoning. The best known work in this area is that of McCarthy [83], which explores a form of reasoning in which propositions and terms are asserted within specific contexts using the relations $ist(c, p)$ (meaning that proposition p is true in context c) and $value(e, c)$ (designating the value of term e in c), and in which *lifting formulas*

are employed to transfer results to more general contexts. The principal goal of this model of reasoning is to allow AI systems that are designed for narrow contexts of use to be later generalised (with appropriate qualifications) to be of broader applicability. McCarthy defines contexts only as abstract objects, and does not address context modelling techniques. Other forms of context-based reasoning include Buvac and Mason's propositional logic of context [84] and Giunchiglia and Ghidini's local model semantics for contextual reasoning [85].

In linguistics and natural language processing, context is used as a tool for disambiguating sentence meaning. Two types of context are considered: the linguistic context, which is determined by surrounding words and sentences, and the non-linguistic or situational context, which includes the identity of the author/speaker, the purpose, the intended audience, and so on [86].

The field of information retrieval aims to identify, and present to the user, documents that are relevant to the current context. The usual approach is to analyse the content of documents viewed by the user, often by computing the frequency of terms that appear in the text, with the assumption that similar documents will also be useful. The extraction of context information only from viewed documents is clearly quite restrictive. Other approaches analyse access patterns to generate profiles of user interests, which can be used in conjunction with the current browsing state in order to support a richer form of context-awareness [87]. Brown and Jones [48, 49, 88] investigate the use of the pervasive computing notion of context to further enhance information retrieval. They describe the use of context information (including sensed data related to parameters such as location and temperature) and context triggers to support interactive (user-driven) and proactive (autonomous, context-driven) document retrieval.

It is clear from this brief survey that each of the described domains has unique requirements in terms of the types of context that are relevant, and the techniques that are appropriate for acquisition and modelling of context information. As a result, the various research topics are largely independent of one another (although there is overlap between the AI fields of machine learning, knowledge acquisition and reasoning), and are likely to remain so in the future. The research presented in this thesis makes no attempt to align itself with these parallel investigations of context. However, as demonstrated by Brown and Jones, the problem of information retrieval represents an interesting application area to which the pervasive computing notion context-awareness can be applied.

Chapter 3

Related work

This chapter presents a discussion and critical evaluation of a variety of tools that can be used to simplify the construction of context-aware pervasive computing applications. It covers software infrastructures and architectures, context modelling techniques, programming abstractions and toolkits, as well as software design approaches. Where relevant, the survey assesses the degree to which the solutions accommodate the types of context information identified in Section 2.2, including uncertain information, histories and dependencies. It also highlights the principal areas in which the solutions fall short of requirements, motivating the research presented in the following chapters.

The survey is organised as follows. Section 3.1 focuses on toolkits and infrastructures that have been proposed to facilitate the acquisition of context information from sensors. Section 3.2 characterises a variety of high-level context modelling solutions, together with their supporting software infrastructures, which enable applications to formulate queries about the context. Section 3.3 explores a small set of programming abstractions that have been developed to simplify the implementation of context-aware software, while Section 3.4 presents some tools that assist with design tasks, including the choice of appropriate types of sensor and the exploration and prototyping of rule-based context-sensitive behaviour. Section 3.5 characterises a broad set of preference modelling solutions, and briefly analyses their applicability to context-aware software. Finally, Section 3.6 summarises the current state-of-the-art in the field of context-awareness, highlights a variety of shortcomings, and positions the contributions of this thesis in relation to these.

3.1 Context gathering infrastructure

3.1.1 Overview

The derivation of useful, high-level context information from sensors is one of the most fundamental problems in the development of context-aware software; consequently, this was a key focus of much of the early research in the field. Over the past decade, the focus has shifted gradually from the abstraction of information from simple homogeneous sensing infrastructures (which more often than not were concerned with location data [2, 6, 41, 89–94]), to more general techniques for extracting rich types of information from diverse (and dynamically changing) collections of sensors. This survey focuses on solutions of the latter type.

The Context Toolkit, developed by Dey et al. [21, 95], provides programmatic support for the distributed acquisition of context information from sensors, in the form of the context component abstraction. The toolkit defines the following abstract component types [95]:

- *widgets*, which function as software wrappers for sensors;
- *interpreters*, which raise the level of abstraction of context information to better match application requirements (for example, transforming raw location coordinates to a building and room number); and
- *aggregators*, which combine different types of context information related to a single entity.

By drawing upon standard libraries of reusable components that instantiate these three abstract types, programmers can easily incorporate context gathering functionality into applications to enable context-aware behaviour. The ability to connect components into complex arrangements gives the component model considerable flexibility and power. The utility of the model is also demonstrated by the array of prototypical applications that have been built using the toolkit; these include a simple location-awareness tool called the In/Out Board [21], a context-aware intercom for the home [11] and a reminder tool [16].

Widgets incorporate historical context information using relational databases for persistent storage, and implement an information model, based on simple attributes, that allows constant values to be represented alongside sensed data. However, profiled information is not supported, and the representation of uncertain context information is not explicitly addressed (although quality information can be captured by additional attributes, if required).

Gray and Salber [96] propose a refinement of the context component abstraction that provides improved support for imperfect context information. They replace the

three abstract component types with a single generic type that, in addition to standard data input and output channels, incorporates control, action and metadata channels. The control channel is used to support dynamic reconfiguration of components, while the action channel performs required operations on the environment (for example, using actuators). Metadata channels carry relevant information about the quality and source of the context information.

Schmidt et al. [23, 97, 98] describe a simple layered architecture which they employ to synthesise context information from a heterogeneous set of sensors. As their architecture is designed for use on autonomous “smart objects” that have been instrumented with multiple sensors, such as next-generation mobile telephones, distribution of context information is not considered. The architecture is structured to support a two-phased interpretation process. The first phase maps the output from each sensor into a variety of cues. These are features derived from a set of sensor readings, such as the average and standard deviation. The second step combines the cues to form an abstract context description, using rule-based algorithms, statistical methods, neural networks [98, 99], or other interpretation techniques.

The uncertain nature of context information is reflected within the context model that is used in conjunction with the architecture. This describes the environment as a vector of pairs, where each pair consists of a value together with a probability estimate. It is also accommodated by the scripting primitives that accompany the model [97]; these allow actions to be automatically invoked when a context event is detected with a predefined probability.

The simplicity of the architecture (and also of the associated context model and scripting primitives), and its focus on sensed data only, prevents it from supporting historical, static and profiled context information. Similarly, its support for uncertainty is limited, as quality can only be characterised through probability estimates, while unknown and ambiguous context information cannot be adequately captured.

Finally, Chen and Kotz [22, 100, 101] propose an infrastructural approach to context acquisition, named Solar. This supports the sharing of context gathering infrastructure, crucial in complex, large-scale pervasive systems, in a manner that is not possible with the previously described solutions. The infrastructure is based on the use of a graph abstraction to specify connections between components. The graph nodes are either *sources*, which represent sensors, or *operators*, which are processing components that perform interpretation and aggregation. Context information traverses the components in the form of event streams.

The graph abstraction is used as follows. Applications produce textual specifications of their context requirements in the form of graphs. Solar uses these descriptions to create the required operators (sharing these where possible with other applications) and event subscriptions. As new operators can be dynamically inserted as required, and subscriptions can be context-dependent (for example, changing in

accordance with the user's current location), Solar supports a more flexible form of context gathering than the Context Toolkit and the architecture of Schmidt et al.

The main disadvantage of Solar, however, is its relatively simple context model. Unlike the Context Toolkit's widgets, Solar's components do not, by default, maintain persistent historical data, nor do they support static or profiled context information. Similarly, the context model lacks the explicit support for quality estimates that is provided by the solution of Schmidt et al.

3.1.2 Discussion

A powerful yet generic context sensing infrastructure that can be shared amongst a variety of applications greatly reduces the complexity associated with constructing context-aware applications, as shown by Dey et al. [11, 16, 21, 95]. However, such infrastructures have three key limitations. First, their emphasis on sensed context naturally biases them against providing strong support for static and profiled information. Second, their usual construction as distributed components, each concerned with only one type of context information, tends to lead to a fragmented context model. This makes it difficult or impossible to capture relationships and interdependencies, and to support context management tasks that span different types of context (e.g., conflict detection and resolution). Third, applications that use a rich set of context information are usually required to interact with multiple components of the sensing infrastructure in order to obtain this information.

These problems can be overcome by introducing a layer of separation between the application and the context sensing infrastructure in the form of a *context server*. The server maintains a rich, integrated model of context, performs context management tasks, and provides applications with a single interface through which they can request context information. The following section surveys a variety of proposed context servers, focusing on the features of their respective context modelling approaches.

3.2 Context models and servers

3.2.1 Overview

Schilit et al. [26, 102] pioneered the development of infrastructural support for context-aware computing a decade ago, by proposing an architecture comprised of distributed context servers called *environment servers* and *user agents*. The architecture partitions the context description amongst the servers and agents, which store context information in the form of simple environment variables. Environment servers maintain information related to domains such as rooms, project groups or other logical or physical entities, while user agents record context information for

a single user. Both servers and agents allow context-aware applications to retrieve context information synchronously using remote procedure calls, or asynchronously by subscribing for notifications.

The simplicity of the context model used by this architecture makes it unsuitable for capturing historical and uncertain context information, as well as relationships and dependencies between different classes of context. Subsequent proposals have adopted successively more sophisticated context models.

The Cooltown model [27] embeds context information within a Web-based framework, associating each entity (a person, place or thing) with a description retrievable via URL. A simple location-based discovery mechanism serves as a form of bootstrapping for context retrieval. This involves the use of a wireless beacon to transmit the URL of the local environment. From this, the other people, devices and objects present in the environment can be discovered. The context model is very informal, as arbitrary information can be embedded within the Web pages. This feature, together with the restrictive discovery mechanism (which is based on the assumption that only information about the local environment is required at any time), limits the utility of the model. Specifically, Cooltown is suitable as an awareness and information retrieval tool (useful for applications such as museum and tour guides), but not as a more general platform for building context-aware applications.

Harter et al. [2] adopt a more formal, object-oriented context modelling approach, and propose a three-layered context management architecture. At the lowest layer, context information resides within a relational database. The middle layer augments the functionality of the database, providing (i) support for the object model (using CORBA object proxies to allow remote queries), (ii) caching of context information to improve efficiency, (iii) direct routing of frequently changing context information to applications (bypassing the database) and (iv) location-specific services, including support for containment queries and spatial monitoring. The upper layer incorporates the clients; these are context-aware applications and producers of context information, including sensors and resource monitors.

The object-based context model employed in this architecture is formal and expressive, and combines both sensed and non-sensed information into an integrated world model. However, it provides no explicit support for histories or uncertain context information.

Pascoe [103] describes a Contextual Information Service (CIS) that provides a shared repository of context information for wearable computing applications. Like the architecture of Harter et al., this supports an object model that integrates sensed information with additional types of context information. Pascoe terms his objects *artifacts*; these possess a name, type and set of states. The states are described in a state catalog, and have one or more formats and types, methods that translate between these, and operations. Relationships between artifacts can also be

represented; these are either explicitly defined by the user, or inferred by the CIS.

Each artifact state has an associated monitor that manages the corresponding sensing components to achieve the required quality of service, and maintains historical information to improve accuracy and provide continued estimates in cases of sensor failure. Unfortunately, the quality and historical information are not mapped into the artifacts, where they could be exploited by applications. Moreover, the representation of ambiguous or unknown context information is not addressed by the model.

Judd and Steenkiste [104] apply a more fragmented approach than the previous solutions, in that they assign each type of context information to a distinct contextual service. In order to simplify the task faced by applications in interacting with these services, they propose the use of a generic service interface. The query facility offered by the interface is relatively powerful, allowing the client to place a time limit on the response and specify bounds on *meta-attributes*, including accuracy, confidence, update time and sample interval. The data model used internally within the services is not specified; this is implementation-dependent, and related to the type and characteristics of the context information.

There are several disadvantages to this approach. First, clients are required to explicitly specify the names of services in queries; as the set of services may be large and dynamically evolving, this places a considerable burden on the client to perform service discovery. Second, the highly fragmented view of context that results from the use of many separate services requires clients to carry out multiple steps to satisfy all but the simplest of queries (as shown by the examples in [104]), and leads to a clumsy solution for representing relationships¹. It also complicates context management tasks, such as detection and resolution of conflicts arising between different classes of information.

Further solutions have been proposed by Hull et al. [25], in form of their SitComp service, by Salber and Abowd [74], in the form of a generic context server, and by Ebling et al. [24, 105], in the form of their Owl context service. However, as detailed descriptions of these services and their context models have not been published, they will not be discussed further.

A variety of services dealing specifically with location information have also been developed [6, 18, 90, 106–108]. As the properties of location data are better understood than those of context information as a whole, these often provide more sophisticated management functionality, such as conflict resolution [106, 107] and privacy support [107, 108]. These services can be employed as suppliers of location information to more general context services such as those discussed above.

¹Judd and Steenkiste propose the use of additional services to capture these.

3.2.2 Discussion

The information models implemented by these context services exhibit various limitations. Several (but most notably the Cooltown model) lack the formal basis that is required in order to capture context information in an unambiguous way and to support reasoning about its properties. The attribute-based model employed by Schilit et al. within their servers and user agents is unsuited to capturing historical information; the object-based models are less restrictive, but provide no special support for the modelling of histories. Most of the solutions allow relationships to be expressed, but all fail to provide any special support for the modelling of dependencies or for related context management tasks. Finally, all of the models fail to adequately address uncertainty; some allow quality metadata to be expressed, but the modelling of ambiguous information (e.g., “the location of x is y or z ”) and the explicit modelling of unknowns are both overlooked. This last problem is of crucial importance, as effective reasoning is impossible without the ability to distinguish between information that is absent from the context server because it is false, unknown or uncertain.

A further problem, common to all of the surveyed solutions, is a lack of context management functionality. Beyond their main roles as repositories of context information and query engines (supporting synchronous queries and asynchronous notification of context changes), the servers implement very little additional functionality. The solution of Harter et al. provides special treatment for spatial data, supporting spatial monitoring and advanced location-based queries; however, neither this context service, nor any other surveyed, offers the sophisticated conflict resolution, enforcement of integrity constraints, or support for privacy policies that are increasingly common within location management systems.

3.3 Programming abstractions for context-aware computing

3.3.1 Overview

The software infrastructures described in the previous sections remove many of the challenges associated with the acquisition, storage and dissemination of context information, thereby greatly simplifying the task faced by the developer of context-aware applications. However, there is a growing consensus that the abstractions provided by context services (specifically, their context models and query interfaces) are not the best ones with which to program highly flexible and robust context-aware behaviour. In light of this trend, a variety of high-level programming models and abstractions, tailored to the needs of context-aware software, have recently emerged.

These are the subject of the remainder of this section.

As described in Section 1.4, *triggering* has been widely used as a programming model for adaptive software systems. This is an event-based model in which actions are automatically invoked upon the detection of relevant changes in the environment. As this solution is not specific to context-aware systems, a comprehensive treatment falls outside the scope of this survey. Instead, two loose derivatives of the model that provide extensions to meet specialised needs of context-aware applications - the stick-e note model and a context-sensitive object model - are characterised, while further discussion of the triggering model is deferred until Chapter 5.

The stick-e note framework, proposed by Brown [29, 70], was originally created to simplify the development of a narrow class of applications involving the presentation of information to users in a context-dependent fashion. However, it has since found an additional use within note-taking applications, allowing notes to be automatically tagged with metadata describing the current context. The framework encompasses not only a model for associating information (termed *stick-e notes*) with contexts, but also a triggering engine that selects and presents the notes to users in a context-aware manner. Its goal is to reduce the task of constructing the aforementioned application classes to a document authoring problem, rather than a software engineering one.

Each stick-e note is specified using SGML (a tag-based markup language related to XML), and has both content and an associated context. As the format is extensible, potentially rich context descriptions can be produced. The utility of the stick-e model has been demonstrated by its use in implementing several applications, including a tourist guide [70] and a tool that supports the recording of field observations by ecologists [103, 109]. However, the model is clearly not applicable to most types of context-aware application.

Yau et al. [31] propose a model that applies triggering at the object level. This model separates application objects into a context-sensitive interface and a context-independent implementation. The interface effectively consists of a set of triggers that associate context events with actions on the object implementation. This is compiled to produce a container that transparently gathers the required context information and invokes triggers in response to relevant context changes. The context model used in the specification of interfaces is unsophisticated, being based on simple variables. Uncertainty, relationships and historical context information are not addressed.

The model is designed principally to support dynamic reconfiguration, and one of the motivating applications that Yau et al. consider centres around the establishment of spontaneous interactions between users when in close proximity with one another (for example, between students and instructors in classroom settings). Like the stick-e note model, the context-sensitive object model supports only limited

classes of context-aware behaviour. It is not suited to domains such as information retrieval [48, 110] or the context-aware communication scenario considered in this thesis (see Section 1.5), in which actions involving the use of context information can be initiated by users rather than in response to context changes.

In addition to these two application models, logic-based abstractions of context that simplify the task of describing context-aware behaviour have been proposed independently by Dey et al. [16, 111] and Ranganathan et al. [10, 112]. These were both developed alongside specific context-aware applications (a reminder tool called CybreMinder in the case of the former, and a chat program called ConChat in case of the latter); however, their applicability is much broader.

The situation-based model of Dey et al. is the simpler of the two. It is founded on the observation that the context component abstraction of the Context Toolkit (described in Section 3.1) is not the right abstraction for all tasks. The situation model that Dey et al. propose in order to overcome this problem allows programmers (and users) to define high-level contexts (i.e., situations) in terms of conditions upon one or more attributes, without having to specify the interactions with the widgets, interpreters and aggregators that collect the required types of context information.

Situations are formed from a list of equalities (e.g., “location = room 383”) and inequalities (e.g., “price > 65”), implicitly connected by logical conjunction. This model is very restrictive, as it is unable to express alternatives (e.g., “location = room 383 or location = room 384”), relationships (“ x is near y ”) or uncertainty (“location is possibly room 383”). A large class of expressions involving quantification (“there is a person whose location is room 383” or “all of the members of the project group are present in room 383”) are also excluded.

The model of Ranganathan et al. overcomes some of these limitations. It captures context information in terms of predicates of the form:

$$\text{Context}(\langle \text{ContextType} \rangle, \langle \text{Subject} \rangle, \langle \text{Relater} \rangle, \langle \text{Object} \rangle)$$

As the relater is not restricted to the equality and inequality operators, arbitrary binary relationships can be expressed, such as:

- $\text{Context}(\text{location}, \text{Emma}, \text{entering}, \text{room } 383)$; and
- $\text{Context}(\text{lighting}, \text{room } 383, \text{is}, \text{off})$.

Unfortunately, ternary and higher arity relationships cannot be accommodated by this model. This implies that historical context information (e.g., “at 10:35am the location of Emma is room 383”) and certain types of uncertainty (“location of Emma is room 383 with probability > 0.8”) have no straightforward representation. The treatment of unknown or ambiguous information is additionally not addressed.

Context predicates can be combined using conjunction, disjunction and negation, and parameterised using universal and existential quantification. In order to ensure

that the evaluation of an expression involving quantification always terminates, the model applies domain restriction. That is, the value of a quantified variable is bounded by the set of objects of the relevant type residing within the system. This approach can be very inefficient when the set of objects is large (which will often be the case in complex pervasive systems), or when quantification occurs over several variables².

3.3.2 Discussion

The investigation of programming tools and abstractions for context-aware systems has largely taken a back seat to the development of infrastructural support. Consequently, there are many gaps that remain to be filled, and a variety of limitations exhibited by the few solutions that have been proposed in this area.

The two trigger-based programming models that were presented are each tailored to narrow classes of context-aware applications. As a result, there is a need for models that support alternative types of behaviour for which triggering is not appropriate, such as the use of context information in the execution of user-initiated tasks. In addition, there is considerable scope for the investigation of models of triggering that account for special characteristics and limitations of context information, including historical data and uncertainty.

The logic-based context abstractions have shown great promise, serving as useful formalisms for describing contexts in high-level terms. Their utility has already been demonstrated with respect to two distinct application types (namely, reminder tools and chat programs), but is potentially much broader. Amongst the benefits of these formal abstractions (and particularly the more powerful predicate-based model of Ranganathan et al.) are the ability to:

- perform logical reasoning about contexts; and
- meaningfully combine contexts using the logical operators in a manner not possible with the context models described in Section 3.2, thereby deriving increasingly complex context descriptions.

The two applications that were explored in conjunction with the models - Cybre-Minder and ConChat - merely scratched the surface in terms of these benefits. In order to fully realise the potential of these abstractions, refinements are required in order to support richer types of context information (including relationships, dependencies and historical information), and to accommodate uncertainty.

There is also a requirement for improved understanding of suitable development methodologies that can be applied in conjunction with these models. To date, very

²However, Ranganathan et al. do not make it clear whether quantification over multiple variables is permitted; the examples they provide all involve the quantification of a single variable over a single context predicate.

little emphasis has been placed on this issue. Some researchers have enumerated the design and implementation tasks involved in developing software using their models (such as Dey, in relation to his Context Toolkit [95]); however, the tasks are almost always described informally, and without adequate guidance in terms of the process or complex issues that are involved.

3.4 Design tools for context-aware software

3.4.1 Overview

The development of tools that support a structured exploration and specification of the design requirements of context-aware software has lagged even further behind that of programming models. This section describes partial solutions that have recently emerged to address two distinct design tasks, while Section 3.4.2 discusses the potential benefits of further formal design tools and methodologies.

Gray and Salber [96] present a design framework that supports an informal exploration of the types and characteristics of context information required by an application, focusing particularly on quality of service (QoS) needs. The exploratory phase forms a precursor to the identification of suitable sensors, and the design and implementation of a context sensing infrastructure using Gray and Salber's variant of the Context Toolkit, which was introduced previously in Section 3.1.

The framework identifies a set of design parameters (termed context meta-information) that are applicable to each type of context. These include representation, quality attributes, sensory source, interpretation steps and forms of actuation. The design process, which Gray and Salber illustrate by example using a museum application, follows a checklist approach. This supports the designer by serving to “identify issues, make issue coverage visible, assist in the documentation of the design process and offer a means of traceability in design decisions”³.

The framework is very informal, and, according to Gray and Salber, requires further refinement and validation using a broader set of design case studies. It is appropriate only for sensed context information, and does not explicitly explore requirements in relation to historical data, which are also very pertinent at the design stage.

Sohn and Dey [113] address the issue of designing appropriate context-aware application behaviour that provides a close fit with user requirements. They present a design tool, called iCAP, which supports rapid prototyping by allowing users to easily specify and experiment with rule-based behaviour in a graphical environment. iCAP depicts the application's context inputs and outputs, and allows users to manipulate the rules that specify the relationships between these. This is useful both in enabling

³ [96], page 328.

the application designer to quickly identify appropriate application behaviour, and in allowing the user to customise the application's functionality at run-time. While promising, the tool remains an early prototype. The context model is very simple, with no support for historical or uncertain information. The conditions that are used to form rules take the form of conjunctions of simple equalities and inequalities; these essentially conform to Dey's situation abstraction, which was introduced in the previous section. Moreover, the design of a realistically-sized context-aware application using the tool does not yet appear to have been attempted.

3.4.2 Discussion

The two design tools described together address only narrow subset of the design tasks associated with context-aware software. There is consequently a need for a better understanding of the design process as a whole, and for developing methodologies and tools for tasks such as:

- the exploration and specification of *all* types of required context information (not just sensed information, as in the design framework of Gray and Salber);
- the design of suitable privacy policies, and application functionality that conforms to these, in order to protect users' personal context information; and
- the design of user interfaces that address the challenges of pervasive computing environments, and achieve an appropriate balance between application autonomy and user control.

The design approaches should be consistent with, and offer a straightforward mapping to, implementation tools. For example, Coutaz and Rey motivate the need for an operational model of context “that can be used and refined at every step of the development process” [114]. This philosophy is reflected in the modelling solutions presented in later chapters of this thesis.

3.5 Preference and user modelling

3.5.1 Overview

There is growing recognition of common usability challenges associated with context-aware software related largely to a lack of transparency of, and user control over, applications' actions [115–117], and a consequent need for improved techniques for eliciting and capturing user requirements and preferences [118–122]. To date, however, there has been very little research addressing these issues.

Byun and Cheverst [118, 119] explore the integration of user modelling techniques developed in AI with context-awareness. Specifically, they exploit probabilistic learning techniques in order to automatically elicit user requirements. This work appears promising; however, explicit means of representing user preferences are also required. The use of an explicit representation allows users to formulate their own preferences if desired, and also provides a tool for exposing preference information and thereby providing transparency, so that users are able to understand the motivation for their applications' actions, and to make corrections as necessary. This approach can be compatible with automated learning techniques, as shown later in this section and again in Chapter 5.

The utility of user preference information has been demonstrated by context-aware applications such as Active Messenger [123], which captures user preferences in relation to the use of communication channels within a configuration file. However, there is a lack of sufficiently formal and generic approaches to modelling context-dependent preferences; currently, preference formats are tightly coupled to specific applications, inhibiting reuse and complicating the task of the user in specifying preference information.

Some researchers view preferences as a special type of context, modelling these in the same manner as other types of context information. This approach is adopted by the CC/PP standard developed by the W3C for the exchange of context and preference information in Web-based applications [124, 125]. In this solution, context and preferences are both represented as attributes using the Resource Description Framework (RDF) [126]. This solution is suitable for expressing very simple requirements (for example, the set of languages that are acceptable for presenting information to a given user), but not for more sophisticated, context-dependent preferences (such as those required by Active Messenger to enable appropriate choice of communication channels).

As there is currently no adequate preference modelling solution available within the context-awareness domain, the remainder of this section explores some of the solutions employed in other fields and analyses the suitability of these for use in context-aware software.

In decision theory, preferences are commonly represented as utility functions that map attributes such as cost, reliability or performance to a corresponding numerical measure of utility [127, 128]. Expected utility theory is a variant that covers decisions in which choices have uncertain outcomes⁴. Decision-making using these preference models can be viewed as an optimisation problem that seeks to maximise the utility of the choice.

One of the problems of this approach is the difficulty associated with constructing

⁴Unfortunately, it cannot deal with uncertain inputs, as required of a preference model for specifying context-dependent preferences over uncertain context information.

an appropriate utility function, particularly when there are many attributes involved. This makes the approach unsuitable for use in complex context-aware systems in which preferences must be specified over rich (and dynamically changing) contexts.

Agrawal and Wimmers [129] propose an alternative preference model that does not require the construction of a single preference function that reflects all of the user's requirements, but instead allows many separate preferences to be specified, which can be later combined to support decision making.

Unlike utility functions, the preference functions proposed by Agrawal and Wimmers are applied to tuples of values (which need not be numerical). The functions produce scores that are values in the set $[0, 1] \cup \{\dagger, \perp\}$, where the numerical scores capture relative desirability (such that higher scores indicate greater desirability), the special score \dagger represents a veto, and \perp represents indifference (that is, an absence of preference).

The simplicity and ease with which separate preferences can be combined using this approach makes it better suited for use in context-aware systems than the utility functions of decision theory. However, several shortcomings remain, including a lack of support for flexible, context-dependent preferences and an inability to cope with the uncertainty that arises when making choices over imperfect context information.

The decision theoretic approach and the preference model of Agrawal and Wimmers, which both assign scores or ratings that impose an implicit ordering upon a set of choices, are known as quantitative approaches. An alternative representation of preferences is provided by the qualitative approach, in which an ordering amongst choices is captured directly, often in the form of binary preference relations. Chomicki [130] adopts this solution in order to rank results of database queries according to user preferences. In his approach, users specify preferences over tuples using first order logic. A preference formula, labelled C , defines a preference relation as follows: t_1 is preferred to t_2 if and only if $C(t_1, t_2)$.

Chomicki uses a preference relation to prune the tuples presented to the user by removing any tuple that is *dominated* (that is, any tuple for which there is at least one other tuple that is preferable according to the relation). This pruning process is not entirely satisfactory, as an empty set of tuples can result when the relation does not produce a strict partial order on the tuples. Preference relations can be combined using the set operators, union, intersection and set difference. However, the problem of empty results becomes even more prevalent when using combined preferences, as conflicting preferences often induce cycles in the resulting relation.

Other approaches to evaluating and combining qualitative preferences are possible that do not suffer from the same shortcomings; however, these have other limitations. Cohen et al. [131] consider the construction of a total ordering of a set of choices that best agrees with a qualitative preference function. They show that, in general, this problem is NP-complete, but an approximately optimal ordering can

be constructed easily using a greedy algorithm. Their algorithm uses a more general preference model than that employed by Chomicki, in which a preference function, $PREF(u, v)$, maps pairs of choices to values in the range $[0,1]$ (rather than booleans) that reflect the certainty with which choice u should be preferred to choice v .

Cohen et al. combine a set of n preferences by computing a linear function of the form $PREF(u, v) = \sum_{i=1}^n w_i R_i(u, v)$, where $R_i(u, v)$ is the rank ordering produced by the i th preference function and w_i is a weight assigned to this ordering. They also present a learning technique by which an appropriate linear composition can be constructed over a series of iterations, given the set of rank orderings and appropriate feedback at each iteration.

However, this approach is relatively inefficient (even when the greedy algorithm is used for preference evaluation over a set of choices), and is considerably more complex than the preference modelling solution of Agrawal and Wimmers. Additionally, like the other solutions, it does not provide a natural way to deal with context-dependent preferences, nor to accommodate uncertain inputs.

Other preference representations do not rely on functions (or relations) at all. The Vector Space Model is an alternative approach that evaluates choices according to similarity between vectors. This model is primarily of interest in problems involving the ranking of documents based on their text, such as those experienced in information retrieval and filtering applications. The Vector Space Model is typically applied to these problems as follows. Documents and user preferences are each represented by n -dimensional vectors, where each dimension corresponds to a relevant term or concept. In order to determine the relevance of a document, the similarity between the document vector and the preference vector is computed by measuring the cosine of the angle between the two vectors.

Çetintemel et al. [132] demonstrate the use of the model for push-based dissemination of Web pages. They represent user preferences as a collection of interest vectors, where each vector corresponds to a cluster of similar documents that are relevant to the user. User feedback is exploited to enable fine-tuning and evolution of the interest vectors over time.

The Vector Space Model is not well suited to describing context-dependent requirements, owing both to the difficulties associated with describing contexts in vector form, and the inability of the model to accommodate uncertainty.

Less formal preference models, which lack the mathematical basis shared by the quantitative, qualitative and vector-based models, are also possible. One example is P3P's preference language, APPEL [133], a standard proposed by the W3C for expressing privacy preferences. P3P [134] and APPEL together allow Web sites to express their privacy policies, and users to capture their preferences, such that decision-making about the sites to which users release their information can be automated. Policies and preferences are expressed in standard vocabularies, using

XML.

Solutions of this type are generally applicable only in narrow domains, do not easily allow preferences to be combined (particularly when these are conflicting), and rely on the existence of well-defined preference vocabularies. Currently, there is no standard vocabulary that is sufficiently general to capture the rich and dynamic requirements of users in relation to context-aware software.

3.5.2 Discussion

Although there are many existing preference modelling solutions, there is currently no adequate, and sufficiently general, technique for modelling the context-dependent preferences that are required for flexible user customisation and control of context-aware software. Consequently, there is a pressing need for research in this area. Numerous challenging issues present themselves, including the need for user-friendly and application-neutral solutions, and for accommodation of the characteristics of context information identified in Section 2.2, including uncertainty.

There is early research underway concerning the exploitation of learning algorithms to construct user models for use in context-aware software. Such algorithms are not incompatible with explicit preference modelling approaches, as demonstrated by the work of Cohen et al. [131] and Çetintemel et al. [132]. The investigation of a solution that supports the specification of context-dependent preferences by users in combination with automated preference elicitation techniques represents an important avenue for future research. A combined solution reduces the burden on the user to provide comprehensive preference information, while at the same time affording a considerable degree of transparency and user control.

3.6 Summary and conclusions

Recent research in context-awareness has largely focused on the development of software infrastructures that perform tasks such as the acquisition of context information from sensors, persistent storage of the information within servers, and dissemination to applications using both synchronous queries and asynchronous notifications. These infrastructures serve an important role, shifting much of the complex functionality from applications onto the middleware, and thereby simplifying the construction of robust applications, even in the face of an evolving sensing infrastructure [135].

However, these solutions exhibit a variety of shortcomings. First, most are founded on oversimplified models of context that fail to fully account for the challenges inherent in obtaining accurate context information (from sensors, users and software agents), and for the requirement of many applications for rich types of

information, including histories and context dependencies. Second, they ignore important context management functions, such as conflict detection and resolution, enforcement of integrity constraints, and implementation of privacy policies. Third, most of the proposed infrastructures focus primarily on the acquisition of information from sensors, or on the issues of storage and dissemination; there is therefore a need for solutions that seamlessly combine these functionalities into a single, comprehensive infrastructure. Fourth, the abstractions provided by the infrastructures for context description and querying are not always the best ones for programming flexible and robust software. The utility of high-level programming models and abstractions is well recognised, but the few proposals that have emerged in this area remain rudimentary and often limited in their applicability. There is an urgent need for new techniques for describing, and programming with, context information in highly abstract terms, as well as solutions for preference modelling and elicitation that enable software to behave autonomously without materially diminishing the user's sense of control.

In addition, infrastructural solutions by themselves support only the implementation phase of the software lifecycle. As the analysis and design tasks associated with context-aware software present novel challenges that are not addressed by traditional engineering methodologies, other modelling techniques and tools are required. Coutaz and Rey [114] propose the use of context models that can be refined incrementally to support the entire software engineering lifecycle. However, recent efforts to develop such a model, and other design tools, have been inadequate.

It is these observations that motivate the research presented in this thesis. The following chapters develop a framework comprised of an integrated set of conceptual modelling techniques, and a corresponding software architecture and infrastructure, that support the developer throughout the software lifecycle. Chapter 4 and the early sections of Chapter 5 address the need for context modelling techniques that capture the rich characteristics of context information described in Section 2.2. Chapter 4 presents a formal, graphical modelling technique that supports the designer in the task of exploring and specifying the context requirements of a context-aware application. In addition, it describes the mapping of the graphical model to a relational representation that is well suited for use in the implementation of a context management system.

Chapter 5 responds to the current lack of programming tools and abstractions suited for use in context-aware systems by proposing a high-level model of context, two complementary programming techniques, and an explicit preference modelling approach that are all founded upon the context model of Chapter 4.

Chapter 6 addresses the need for comprehensive infrastructures for context-aware systems by integrating the modelling solutions of Chapters 4 and 5 into a layered architecture that incorporates context acquisition and management, preference man-

agement and support for the programming models in the form of a programmer's toolkit.

Finally, Chapter 7 demonstrates, by example, the use of the modelling techniques and software infrastructure of Chapters 4 to 6 to support the software lifecycle.

Chapter 4

Context modelling

The previous chapter demonstrated that current context-aware systems are based on models of context that are often implicit and informal, and incapable of capturing all of the characteristics of context that were outlined in Chapter 2, such as temporal and quality aspects. These models are also inappropriate for supporting context-related tasks throughout the software lifecycle, from analysis of the context requirements of an application through to implementation, instantiation of the context model, and beyond. This chapter presents a context modelling approach, termed the Context Modelling Language (CML), that is designed to help overcome these shortcomings. CML offers a modelling notation that is sufficiently abstract to allow developers to explore and capture the context requirements of an application in a natural way, while at the same time being formal enough to support a straightforward mapping to a context management infrastructure that can supply context information to applications at run-time. In order to meet these requirements, CML leverages an approach that is well established in the design of information systems, which recognises several distinct modelling levels (conceptual, logical and physical) that are distinguished by their degree of detail and abstraction. This chapter focuses on the modelling of context at the conceptual and logical levels. The following sections define some of the terminology used in this chapter, outline the motivation behind the development of CML, and then characterise CML's approach to capturing the special features of context information from the conceptual and logical perspectives.

4.1 Terminology

This chapter addresses the context modelling problem from two perspectives. To prevent confusion, this section characterises the relationship between the two viewpoints and defines some common terms used by each.

A *context model* is an abstract yet formal description of relevant aspects of the

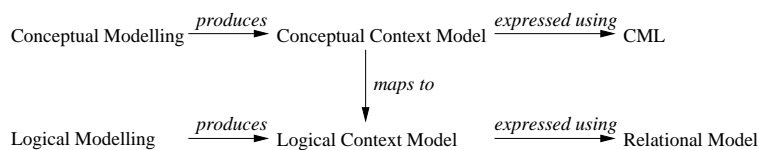


Figure 4.1: Context modelling approach used in this thesis.

real-world context of one or more applications. The latter are sometimes termed *context consumers*, while the aspects of the world that a model reflects are often collectively referred to as the *modelled reality*. A context model can be described in a highly abstract fashion in terms of a *conceptual model* that is designed to be easily understood by people. Alternatively, it can be expressed in terms of a *logical model* that focuses more on the structure and representation of information, and is generally designed to be easily implemented/instantiated. Note that close comparisons can be drawn between the conceptual and logical models of context discussed in this chapter and the conceptual and logical schemas in layered database models such as that described by Halpin [136]. These similarities are natural because the conceptual modelling techniques presented here derive from ORM, a modelling approach created to support the development of conceptual schemas for information systems.

The approach to context modelling proposed in this thesis is illustrated in Figure 4.1. This chapter primarily focuses on modelling at the conceptual level using CML, a novel modelling approach which offers a graphical notation and an accompanying modelling methodology for describing context information. The chapter also briefly describes the modelling of context information at the logical level using the relational model, which has been widely researched and implemented by the information systems community.

A modelling process involving the following steps is assumed:

1. construction of a conceptual model of context requirements using CML;
2. mapping of the conceptual model to the relational model; and
3. generation of a context management infrastructure based on the relational model.

Each of these steps can be at least partially automated. The first step requires substantial input from humans, but can be supported by appropriate modelling tools that assist with the drawing and validation of models. The latter two tasks can be largely automated. However, the development of these forms of tool support falls outside the scope of this thesis. The conceptual models shown in this chapter were produced using simple drawing tools, and hand-mapped to a relational representation. A generic context management infrastructure that can be customised for individual context models has been implemented and is described in Chapter 6.

A context model is populated, or instantiated, at run-time with a set of related information about the modelled reality, referred to as a *context instance*.

4.2 Motivation and approach

As described in the previous chapter, there is currently a lack of understanding of the software engineering tasks associated with context-awareness, and an absence of suitable design tools. This chapter addresses the requirement identified in Section 3.4.2 for suitably general context modelling techniques that support the exploration and specification of an application's context requirements. The context modelling approach developed here is orthogonal and complementary to the informal design framework of Grey and Salber, introduced in Section 3.4.1, which supports design tasks in relation to *sensed* context information, including the identification of QoS requirements and of suitable sensors.

CML enables application developers to construct formal models of context requirements that (i) capture all of the context characteristics described in Chapter 2 and (ii) form a natural basis for the specification of context-aware behaviour using the situation and preference abstractions developed in the following chapter. One of the novelties of CML is that it draws on the wealth of research concerned with information modelling that has been carried out in the field of information systems. There are numerous research topics in information systems that overlap with the context modelling problem. A full survey of these topics falls outside the scope of this thesis; instead, a few of the most relevant are mentioned here to indicate the extent of overlap between the research areas. The conceptual modelling of temporal aspects of information has been widely researched: a survey of some of this work is provided by Gregersen and Jensen [137]. Similarly, the conceptual modelling of information quality aspects is described by Wang et al. [138] and Storey and Wang [139], and the construction of information systems based on sensor-derived information using a combination of concepts from real-time, active and temporal databases is described by Datta [140]. Finally, the problem of incomplete information has been widely studied [141–146]. It must be noted, however, that these areas of research have been largely carried out in an isolated fashion, and that there is consequently no single information modelling approach that is adequate for expressing all of the aspects of context information. This chapter describes a set of integrated context modelling concepts that combines and extends relevant information modelling research. The integration of these concepts into a cohesive model is non-trivial: when considered in isolation, temporal aspects, incompleteness and ambiguity have the potential to lead to complex information models, but when these problems are considered together, there are subtle interactions that must be treated with great care. For example, when historical information is modelled, properties on the information

- such as constraints - assume different semantics depending on whether they are applied to individual snapshots, to entire histories, or to some other level of granularity. The modelling approach presented here is designed as a minimal solution to these problems that on the one hand encompasses all of the key features of context information, and on the other helps to spare the designer from the burden of working with an unduly heavy set of modelling concepts.

Initially, the context modelling concepts were formulated independently of any single information modelling approach, such as UML [147] or the Entity-Relationship model (ER) [148]. This afforded the flexibility to express the concepts in the most natural way. The results of this initial exploration are presented in [40]. Subsequently, a decision was made to reformulate the modelling concepts as extensions to the Object-Role Modeling (ORM) approach [149], developed for the modelling of information systems. ORM was chosen in preference to other modelling approaches (such as the more popular ER model) because of its relative formality and expressiveness, and the close correspondence between its fact/relationship types and the association abstraction that formed the basis for the original modelling approach. One of the particular strengths of ORM, in contrast to both ER and UML, is that it focuses on relationship types as much as on object types, making it easy to specify complex constraints and annotations on relationships. The importance of this feature is apparent in Section 4.4, in which a variety of special relationship types, annotations and constraint types are introduced for context modelling.

The decision to align the modelling concepts with ORM confers several significant benefits. First, it provides the opportunity to exploit a wide variety of constraint types offered by ORM that are important in (but not specific to) the context modelling problem, such as uniqueness, value and subtype constraints. Moreover, by basing CML upon ORM, it is possible to leverage the straightforward mapping from ORM to the extensively implemented relational model. This makes it possible to store context instances within standard relational database systems, enabling extremely efficient querying of context information using widely known database query languages, such as SQL. Finally, ORM has been widely used (mainly within research settings) as a basis for conceptual database design, with the consequence that it offers a mature, well-understood and well-documented modelling approach.

The following section presents a brief overview of ORM, while the remainder of the chapter discusses the extension of ORM to accommodate the key context characteristics enumerated in Chapter 2. The latter discussion focuses first on the construction of abstract context models at the conceptual level, and then on the representational issues of the logical level.

4.3 Introduction to ORM

ORM is a conceptual modelling approach, developed by Nijssen and Halpin [150], that is founded on a data model in which information about objects is embodied in facts. It comprises both a graphical modelling notation and an accompanying design methodology; these are the subjects of the following two sections.

ORM has numerous variants; the following sections describe Formal ORM (FORM), which has been the subject of several books [149–151], and is arguably the most widely adopted and extensively documented variant of ORM.

4.3.1 Modelling concepts

FORM's conceptual schemas comprise the following three components:

- stored fact types;
- constraints; and
- derivation rules.

The stored fact types describe the classes of fact that appear in a database instance. Fact types consist of one or more distinct roles; the number of roles is referred to as the arity. Each role is associated with an object that is characterised by a name and a reference mode, the latter reflecting the database representation of the object (for example, by name or numerical identifier).

Constraints place restrictions on populations of fact types, and reflect the semantics of the modelled reality. FORM supports many types of constraints: two examples are uniqueness constraints (e.g., “each person has only one date of birth”) and mandatory roles (e.g., “every person must have a date of birth”). Uniqueness constraints are classed as internal or external; the former involve one or more roles of a single fact type, whereas the latter span roles of multiple fact types. They effectively capture a type of *functional dependency* that dictates that the combination of objects participating in the roles covered by the constraint is unique over all facts of the given type(s). Stated in a different way, a uniqueness constraint implies that, collectively, the objects participating in the roles of the constraint uniquely determine at most one fact in the case of an internal uniqueness constraint¹, or one combination of n facts in the case of an external uniqueness constraints that spans n fact types. Mandatory role constraints differ in that they govern the participation of individual objects in roles. A mandatory role constraint on an object type O and role r implies that an object of type O that participates in any role of any fact type must necessarily participate at least once in r .

¹This is analogous to the key constraint of the relational model.

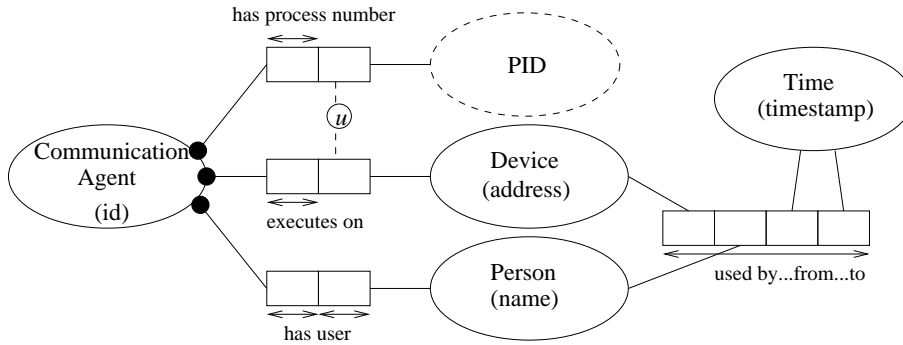


Figure 4.2: FORM modelling example.

Finally, derivation rules describe the production of additional fact types from the base fact types (for example, the average salary of a department from the salaries of individual employees).

Conceptual models are constructed diagrammatically using the notation illustrated in Figure 4.2. Fact types are captured as sequences of role boxes, with each box attached to an object type. Each fact type has a label (e.g., *has process number*, *executes on*, *has user*) that describes its semantics in natural language. Double-headed arrows are placed over roles in a fact type to indicate internal uniqueness constraints, while external uniqueness constraints are indicated by an encircled *u* joined by dotted lines to the roles participating in the constraint. Examples of the former are the two constraints over the roles of the *has user* fact type in Figure 4.2. These imply that each agent has a unique user and each user has a unique agent. An external uniqueness constraint is present between the *has process number* and *executes on* fact types; this captures the constraint that, on a given device, at most one agent is assigned any given process identifier. Mandatory role constraints are captured by placing a dot at the object side of the line that connects the object type to the mandatory role; three examples can be found attached to the *Communication Agent* object type: these indicate that every agent must be associated with a process identifier, a device and a user.

Object types are represented as ellipses containing a name and an optional reference scheme placed within parentheses. Object types that are simple value types (such as the *PID* object type in Figure 4.2, which designates process identifiers) do not require a reference scheme and are shown as dashed ellipses.

FORM also supports a variety of advanced constraint types, including value, subtype, occurrence frequency, ring and relative closure constraints. A discussion of these falls outside the scope of this thesis. For further information, the reader is referred to [149].

Conceptual models, such as that shown in Figure 4.2, describe generic fact types and constraints upon these. An instantiation of a conceptual model populates each

Table 4.1: Example instantiation of the *has process number* fact type shown in Figure 4.2.

<i>Communication Agent</i>	<i>PID</i>
CA03332	9697
CA01204	9834
CA03444	9335

Table 4.2: Example instantiation of the *executes on* fact type shown in Figure 4.2.

<i>Communication Agent</i>	<i>Device</i>
CA03332	130.102.176.36
CA01204	130.102.176.192
CA03444	130.102.176.192

Table 4.3: Example instantiation of the *has user* fact type shown in Figure 4.2.

<i>Communication Agent</i>	<i>Person</i>
CA03332	Michelle Williams
CA01204	Emma May
CA03444	Mark Darcy

Table 4.4: Example instantiation of the *used by...from...to* fact type shown in Figure 4.2.

<i>Device</i>	<i>Person</i>	<i>Start Time</i>	<i>End Time</i>
130.102.176.36	Michelle Williams	Dec 9 10:14:34 2002	Dec 9 14:35:12 2002
130.102.176.36	Emma May	Dec 9 14:56:43 2002	Dec 9 17:32:56 2002
130.102.176.192	Mark Darcy	Dec 9 07:34:01 2002	Dec 9 16:55:32 2002
130.102.176.192	Michelle Williams	Dec 9 15:15:43 2002	Dec 9 15:33:34 2002

of the fact types with a concrete set of facts. Example instantiations of the four fact types shown in Figure 4.2 appear in Tables 4.1 - 4.4.

4.3.2 Modelling methodology

FORM defines not only a modelling notation but also a generic modelling methodology that can be applied to construct conceptual models. This methodology, termed the Conceptual Schema Design Procedure (CSDP) is described in detail by Halpin in [151], and briefly summarised here. The procedure comprises the following seven steps:

- *Step 1.* Identify elementary facts based on familiar examples, and apply quality checks to verify that:
 - objects have appropriate identifiers; and

- facts cannot be decomposed into two or more smaller facts (that is, facts involving fewer roles) without loss of information.
- *Step 2.* Draw the corresponding fact types and apply a population check to ensure that fact types are compatible with examples.
- *Step 3.* Check for pairs of object types that can be combined; this should be done when entities of the two types can be meaningfully compared. Check for arithmetic derivations and ensure that derived information is represented by derivation rules rather than stored fact types.
- *Step 4.* Add internal and external uniqueness constraints. Apply an arity check to fact types, and split fact types whenever this can be done without loss of information. In general, a fact type should be split if it has an internal uniqueness constraint that spans fewer than $n - 1$ roles, where n is the arity of the fact type.
- *Step 5.* Add mandatory role constraints to those roles that must be played by the entire population of the corresponding object type. Check for logical derivations and ensure that information that can be logically derived is represented by derivation rules rather than stored fact types.
- *Step 6.* Add value constraints where relevant to specify the members of a value type, either by enumeration or using regular expressions. Add set comparison constraints to capture relationships between the populations of roles (subset, equality or exclusion). Finally, specify subtyping constraints to capture specialisations of object types.
- *Step 7.* Add other constraints, including occurrence frequency, ring and relative closure constraints. Perform final checks to ensure internal consistency (that each role sequence can be populated in some state), external consistency (that the conceptual schema is not in conflict with requirements and example data), lack of redundancy (that no elementary fact can appear twice) and completeness (that the schema captures all aspects of the requirements).

When modelling small domains, the seven steps are typically carried out once, in sequence. For larger domains, an iterative process may be applied, involving the decomposition of the domain into smaller sub-domains, the modelling of each of these sub-domains independently using the seven CSDP steps, and, finally, the re-integration of the partial schemas into one global schema.

4.4 Context modelling concepts

Having introduced FORM in the previous section, this section describes a set of novel modelling extensions (and associated context management issues) that address the characteristics of context information described in Section 2.2 (heterogeneity, imperfect information, histories and dependencies). The resulting modelling approach is referred to as CML. The extensions are formulated such that upwards compatibility is preserved (that is, that the original syntax and semantics of FORM's modelling constructs are retained).

4.4.1 Classification of fact types

Section 2.2.1 discussed the wide variation that can be present in a set of context information, caused largely by the diverse characteristics of the sources from which the information is derived. This variation manifests itself in differing persistence and error characteristics, with the result that different styles of management are suited to different types of context information. In order to accommodate these, CML distinguishes four classes of context information.

Broadly, context information is classed as static or dynamic. Both types of information can be captured by FORM, but they cannot be differentiated. CML introduces a variety of new graphical notations to overcome this limitation.

Static information represents aspects of the modelled reality that are invariant, such as the type or identifier of a computing device. In CML, this class of information is captured by static fact types. When modelling these, the modeller must specify a set of roles to which the existence of fact instances are tied; collectively, these roles are referred to as the base set. Each role in the base set is annotated with an *s*, as shown in the example of Figure 4.3 (a). Two constraints are implied by this annotation:

- fact instances of a static fact type cannot be modified; and
- a fact instance can only be deleted from a context instance when an object participating in one of its base roles is simultaneously deleted.

Any fact type that is not static is dynamic. Dynamic fact types are further categorised as profiled, sensed or derived, depending on the source of the facts with which the type is populated.

Profiled fact types capture information supplied by users. This class of information is generally slowly changing and reliable. CML denotes profiled fact types with a circular annotation, as shown in Figure 4.3 (b).

Sensed context represents information obtained from hardware or software sensors. An example of this class of context is location information derived from active

bats [2] or GPS devices. Figure 4.3 (c) illustrates the annotation of a fact type as a sensed type.

Typically, sensed context is highly dynamic, and therefore prone to staleness. Additionally, information of this type is often inaccurate, erroneous or unknown as a result of crudeness of the available sensing mechanisms, sensor failures or gaps in the coverage of sensors. CML provides two types of support for these problems which will be described in later sections.

Derived fact types capture information that is computed from one or more other fact types in the context model. This class of fact type is already supported by FORM, and was mentioned briefly in Section 4.3.1. FORM's notation for representing derived fact types is illustrated in Figure 4.3 (d). There are two components to this representation:

- a rule that describes the production of derived facts from other modelled facts; and
- a sequence of role boxes, drawn as for ordinary fact types but marked with an asterisk.

The latter is optional, but in general should be included for clarity and to enable additional information and constraints on the fact type to be represented, such as the temporal, quality or alternative annotations that will be presented in Sections 4.4.2, 4.4.3 and 4.4.4, respectively.

Generally, the derivation rules used in traditional database systems capture straightforward logical or arithmetic derivations. In contrast, context-aware systems frequently rely on more sophisticated and error-prone derivation algorithms that serve to transform low-level information supplied by sensors into more abstract and useful context information. This processing can occur before context information is inserted into a context instance, in which case the resultant information is captured by one or more sensed fact types; alternatively, the base facts can be included in the context model, and the derivation captured by a derived fact type. The choice will usually rest on the utility of the base facts to the consumers of context information. If the level of abstraction of these facts is so low as to render them useless to context-aware applications, they should generally be omitted from the model. In almost all cases, at least some processing of sensor output will be necessary prior to its insertion into a context instance as one or more facts.

The classification of fact types as static, profiled, sensed or derived can support a variety of context management tasks, two of the most prominent being resolution of inconsistency and update management. When inconsistency is detected in a context instance, it can sometimes be resolved by comparing the classifications of the fact types in question. Static fact types can be regarded as highly reliable, profiled

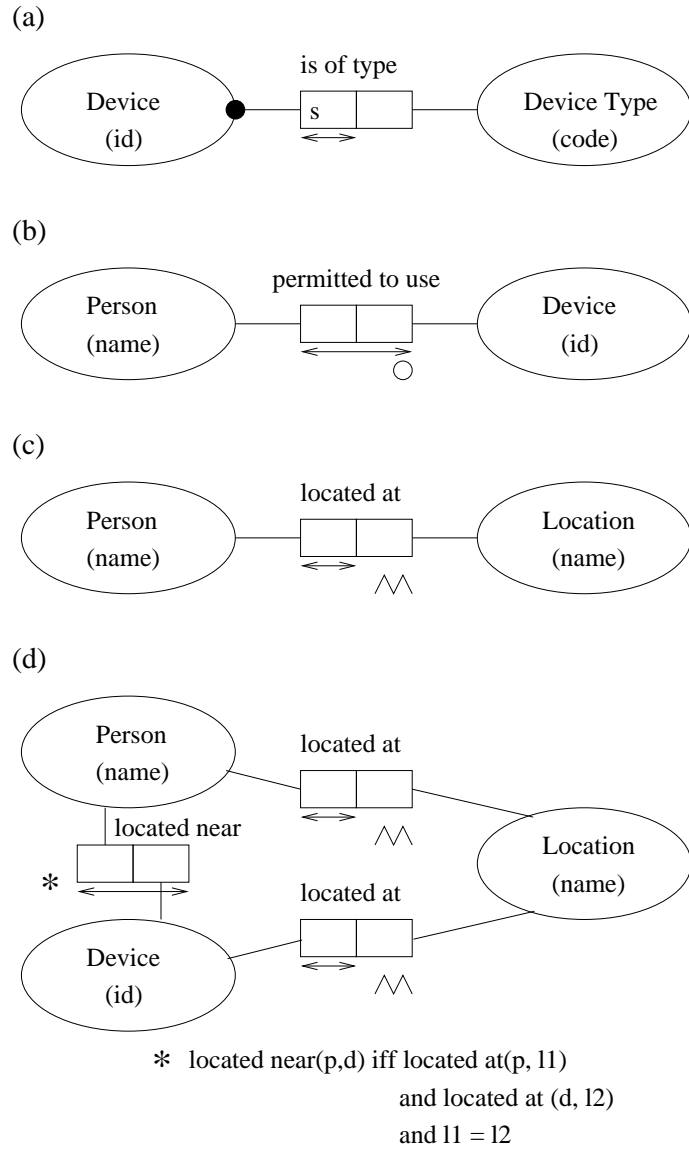


Figure 4.3: Classification examples.

(a) *is of type* is a static fact type for which instances persist as long as the corresponding device objects.

(b) A profiled fact type.

(c) A sensed fact type.

(d) *located near* is a derived fact type. Its instances are computed from those of the two *located at* fact types according to a rule that states that a person p is located near a device d exactly when the recorded location names for p and d are identical.

context somewhat less so, and sensed data generally unreliable and highly prone to staleness. An inconsistency between two of these classes can be resolved by favouring the information belonging to the more reliable class. Inconsistency involving derived fact types cannot occur provided that the data is not explicitly stored, but instead computed on demand, or provided that instances of derived fact types are updated immediately upon changes to the relevant base facts.

When managing updates to context information, static and dynamic fact types require differentiated treatment. Updates to static context must be strictly controlled in accordance with the two constraints described earlier. The rules associated with the three types of dynamic context are less rigid; however, each requires specialised management support. Sensed context poses several challenges. The frequency with which sensed context can change and the scale of pervasive systems imply that highly efficient processing of updates is required. Additionally, support must be provided for the integration of input from multiple sensors into a single fact type, and for processing of the input prior to insertion into a context instance. Profiled context is less problematic, but may require access control to ensure that context information is only accepted from authorised users. Update management for derived context may or may not be required. This depends on whether derived facts are explicitly stored or evaluated dynamically on demand. In many database systems, the latter approach is preferred because it obviates the need to ensure that derived data is kept consistent with the remainder of the database. However, this approach is not well suited to context-aware systems, in which context information generally needs to be obtained and acted upon in a timely fashion, and derivation mechanisms can be complex and computationally intensive. Generally, the most appropriate choice must be determined on a case-by-case basis. FORM offers a notation to differentiate derived fact types that are explicitly stored from those that are computed on demand: the former are marked with a double asterisk, ‘**’, and the latter with a single asterisk, ‘*’.

4.4.2 Temporal fact types

The problem of modelling the temporal aspects of information has been extensively researched within the field of temporal databases. At the conceptual level, this research has mainly focused on extensions to the Entity Relationship model that can capture temporal dimensions in a natural way; a recent survey by Gregersen and Jensen [137] covers numerous such extensions. Other research in the field of temporal databases has focused on temporal extensions to the relational model [152] and temporal query languages [153]. In light of this wealth of research, the goals of this section are twofold. The first goal is to outline the temporal characteristics of context information and relate these to relevant concepts within temporal database

research. The second is to present a set of minimal extensions to FORM that can capture these concepts.

Some temporal databases associate every fact with a temporal annotation (i.e., a timestamp or time interval). This general solution is not required when modelling context; instead, temporal annotations can be restricted to special fact types, referred to henceforth as temporal fact types, which maintain historical information (past, future or both), in addition to current state information. Temporal fact types capture arbitrary time-indexed information, such as schedules for users describing past and planned activities, or histories of a vehicle’s movement over a recent period. The modelling of such histories can be accomplished using the temporal database concept of valid time [154]. Following this approach, a history comprises a set of facts, each of which has an associated valid time that indicates when the fact is true in the modelled reality. Valid times can be either discrete events or continuous time intervals.

Constraints over temporal fact types are considerably more complex than those over non-temporal types. Tautovich [155] differentiates snapshot and lifetime cardinality constraints, and this distinction can also be applied to other constraint types². Snapshot constraints are those constraints that hold when a context description is restricted to a given point in time, t (that is, when temporal fact types whose valid times do not overlap with t are ignored). Conversely, lifetime constraints apply over a complete context description when the valid times of all temporal fact types are disregarded.

CML adopts the convention that all constraints on temporal fact types are, by default, lifetime constraints. This preserves the normal semantics of the constraints when the timestamps that are implicit in temporal fact types are regarded as objects participating in ordinary roles. However, when modelling uniqueness constraints, the lifetime and snapshot interpretations each have useful semantics, so CML supports both. Lifetime uniqueness constraints are used to capture functional dependencies that apply over an entire history of facts. To give an example, a company’s policy that an employee can only hold a given job title once is captured by a lifetime uniqueness constraint which ensures that only a single fact concerning a given employee and a given title appears within a history. Snapshot uniqueness constraints, however, are often more natural for capturing the real-world semantics of relationships between participants in a fact type. For example, when modelling the location history of an object, a uniqueness constraint that states that for each object there is at most one location is a natural constraint that applies at the snapshot level, but not at the lifetime level. Snapshot uniqueness constraints capture temporal depen-

²More recent research has addressed the specification of constraints at finer granularities to allow constraints over intervals of time measured in units such as hours, days or years [156]. These fine-grained constraints are difficult to express and enforce, and fall outside the scope of this thesis.

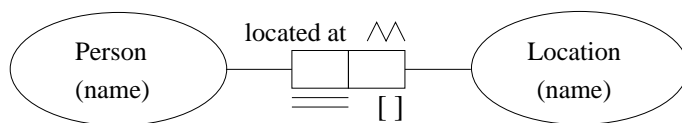


Figure 4.4: A temporal fact type with a snapshot uniqueness constraint that captures the semantics that, at any given time, a person has at most one location.

dencies; these are constraints applied to temporal relations that are satisfied if all snapshots satisfy the corresponding conventional functional dependencies.

The notations CML uses to capture temporal fact types and snapshot constraints are illustrated in Figure 4.4. A temporal fact type is indicated by the '[' annotation: this implies that each fact of the annotated type is associated with a valid time interval $[StartTime, EndTime]$, where the two endpoints are represented by timestamps. Both discrete events and continuous time intervals can be captured (the former by ensuring $StartTime$ and $EndTime$ are equivalent). Internal snapshot uniqueness constraints are indicated, in a similar manner to ordinary uniqueness constraints, by a double line spanning one or more roles on a temporal fact type; this notation is illustrated in Figure 4.4³. The external variant is captured as usual, with the exception that double lines are used to connect the encircled u to the participating roles. For each temporal fact type, at least one snapshot or lifetime uniqueness constraint must be specified⁴.

Care must be taken when combining temporal fact types with other classes of fact type. In general, derived temporal fact types are nonsensical unless at least one of the base fact types involved in the derivation is also temporal. Additionally, some issues associated with combining temporal and alternative fact types are outlined in Section 4.4.4.

4.4.3 Quality

The problem of imperfect context information has been widely acknowledged [6, 23, 24, 72, 96, 104, 157–160], and some of its causes were described in Section 2.2. While measures can be imposed to reduce imperfection, such as consistency checking and resolution, it is infeasible to eliminate the problem altogether, so CML instead adopts measures to minimise its impact. By allowing facts to be associated with quality measures, CML enables the consumers of context information (typically context-aware applications) to make judgements about the extent to which they rely on it.

Common solutions to incorporating support for uncertainty into an information

³Arrowheads can be attached to the extremities of this double line as usual, but are only necessary when the roles participating in the uniqueness constraint are not contiguous

⁴And, as usual, if this constraint spans less than $n - 1$ roles, where n is the arity of the fact type, the fact type must be split.

model can be divided into two classes. The first is the homogeneous approach, which assigns every fact a numerical measure of certainty (or a standard set of such measures, such as accuracy and confidence). This approach has been widely explored in the domains of knowledge representation and reasoning, where certainty is variously represented using probability or possibility measures [161, 162], or certainty factors [163]. It has also been applied by Schmidt et al. [23] within their sensor framework to characterise the likelihood that sensed context accurately reflects reality, and in the contextual services of Judd and Steenkiste [104], in which dynamic context is characterised by the four parameters of accuracy, confidence, update time and sample interval. Finally, within the field of information systems, the association of temporal database information with probability measures has been studied by Biazzo et al. [164]. The homogeneous approach is particularly suited to problems involving logic-based reasoning, and facilitates comparison and composition of certainty measures. In contrast, in the heterogeneous approach, there is no single, standard representation of certainty; instead, each type of information has its own set of quality measures. This approach enables multiple dimensions of information quality to be captured and supports reasoning about quality within a given information type, but inhibits the comparison of quality over different types.

CML adopts the latter solution. The heterogeneous approach presents a more realistic solution than the homogeneous approach for context modelling, given that context information exhibits extremely diverse characteristics and is derived from wide-ranging sources. Rather than imposing a uniform quality model on all types of information and requiring information producers to conform to this model, CML's solution instead enables quality indicators to be matched to the nature of the context information and the capabilities of producers.

CML's quality model is partially inspired by the work of Wang et al. [138, 139], which explores the conceptual modelling of quality requirements within the framework of the Entity-Relationship model. Wang et al. adopt an approach in which attributes, which are analogous to ORM's roles, are tagged explicitly with quality indicators. CML's solution differs in that it regards quality as a property of entire facts; however, both approaches apply a two-tiered solution in which quality is characterised by a combination of abstract quality dimensions (parameters) coupled with concrete interpretations (metrics).

The quality parameters of a fact type are determined by the characteristics of the information embodied in the instances of that type. For example, sensor accuracy and resolution are appropriate quality parameters for sensed information, while the credibility of the person supplying a fact is more relevant for profiled fact types. Some common quality parameters are outlined in Table 4.5, but this list is by no means exhaustive.

Quality metrics provide concrete descriptions of how quality is measured and

Table 4.5: Common quality parameters.

Parameter	Description	Example metrics
Accuracy	The degree of correspondence of measured values to actual values, typically determined by an observer over a number of trials. This parameter is most appropriate for sensed and derived context.	Standard error (for quantitative information), probability of correctness
Confidence	The certainty the producer has in the information supplied. Whereas accuracy provides an objective measure, confidence is subjective. Certainty is particularly relevant to profiled or derived context.	Probability
Freshness	Describes how current information is. This parameter is appropriate for any type of dynamic context, but particularly sensed context.	Production time, time to live
Resolution	Describes the granularity at which the context information is captured. Applies to any type of context.	Frequency of updates, least noticeable difference (for quantitative context)
Credibility	Describes the reliability of the information supplier. Applies to any type of context.	Credibility rating

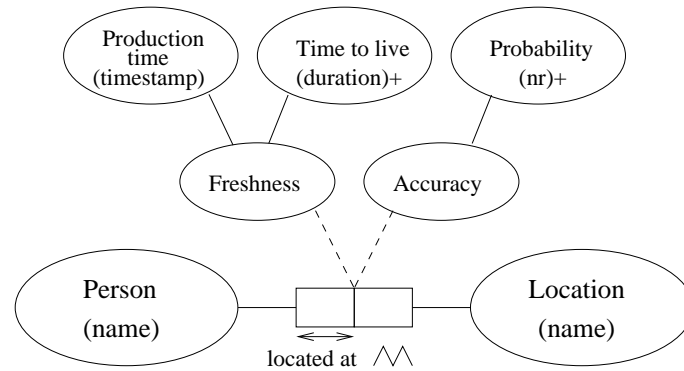


Figure 4.5: Annotation of a fact type with quality indicators.

Table 4.6: Example instantiation of the fact type shown in Figure 4.5.

Fact		Quality measures				
<i>Person</i>	<i>Location</i>	<i>Production Time</i>			<i>TTL</i>	<i>Prob</i>
Michelle Williams	78-123	Mon Dec 9 10:14:44 EST 2002			00:05:00	0.94
Emma May	98-122	Mon Dec 9 10:16:02 EST 2002			00:10:00	0.97
Jane Bennet	78-222	Mon Dec 9 10:13:02 EST 2002			00:15:00	0.92
Mark Darcy	78-123	Mon Dec 9 09:23:13 EST 2002			00:45:00	0.20

represented for a given parameter. A parameter can be associated with several metrics. These should be chosen with the capabilities of the producer in mind; frequently, compromise must be reached between the burden placed on the producer to supply quality information and the requirements of context consumers for an accurate picture of the quality.

The notation CML introduces to capture quality indicators is depicted in Figure 4.5. In general, quality annotations are only attached to dynamic fact types, as static facts are virtually always correct, and quality indicators such as freshness and accuracy are not applicable. Dynamic fact types can be annotated with zero or more quality parameters. In the example in Figure 4.5, the *located at* type has two parameters, freshness and accuracy. Each parameter is in turn associated with one or more metrics, which, like ordinary object types, have a reference scheme shown in parentheses. In our example, the metrics are production time, time to live and probability.

The quality annotations shown in the example imply that each fact that instantiates the *located at* fact type will be associated with three quality values, as shown in Table 4.6.

Quality measures, such as those shown in the table, are treated as first-class information that is inserted, modified and deleted along with the ordinary components of the fact. Therefore, if a fact is derived, its quality measures are likewise produced by the derivation mechanism, often from the quality measures of the base

facts; similarly, the quality measures of sensed and profiled facts are supplied by the sensors (or the corresponding interpreters layered on top of these sensors) and humans who supply the facts, respectively.

As well as being updated in the usual way, changes to a fact's quality measures can be triggered in response to a change in one of the facts upon which it depends. This will be covered in Section 4.4.5.

4.4.4 Alternatives

The second component of CML's support for uncertainty is the alternative modelling construct. This targets cases in which producers of context information (particularly sensors) supply multiple, conflicting pieces of context information. If these cannot be resolved unambiguously, the conflicting information is stored as a set of alternative facts.

The semantics of the alternative construct are as follows: an alternative set of facts, $\{f_1, \dots, f_n\}$, is regarded as accurate (that is, faithful to the modelled reality) if exactly one fact f_i in $\{f_1, \dots, f_n\}$ holds in the modelled reality.

It should be noted that it is not the purpose of the alternative construct to capture multiple representations of non-conflicting information (e.g., location information at different levels of granularity). Alternative representations are instead captured by multiple fact types (e.g., by separate *has GPS coordinates* and *has building number* fact types in the case of location information). This approach ensures that each role in each fact types has a single, well-defined representation or reference mode.

To indicate that alternatives can be present in the instantiation of a given dynamic fact type, the alternative annotation, 'a', is placed beside the type. Note that this annotation cannot be applied to static fact types. Each alternative fact type has exactly one *alternative role* which can take on a set of alternative values, and an *alternative uniqueness constraint* that spans all roles of the fact type with the exception of the alternative role. Facts are regarded as alternatives if and only if the objects participating in each of the roles covered by the alternative uniqueness constraint are identical.

The alternative uniqueness constraint of an alternative fact type resembles an ordinary uniqueness constraint, but takes the form of a dashed, rather than solid, arrow. An example of this notation appears in Figure 4.6, which extends the quality example of the previous section to allow each person to have more than one recorded location.

An example population of the *located at* fact type is illustrated in Table 4.7, with quality measures omitted for simplicity. The alternative uniqueness constraint over the *Person* role in Figure 4.6 implies that facts are alternatives when they pertain

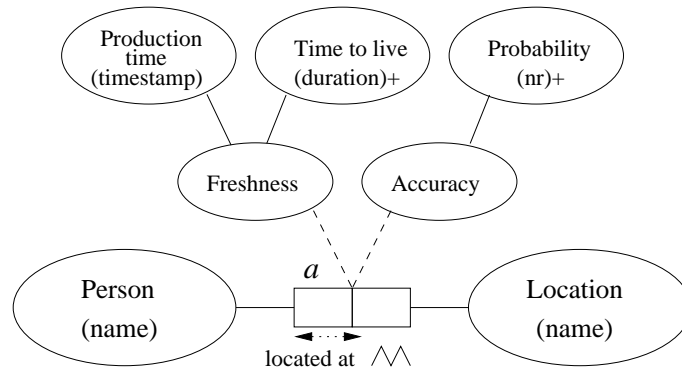


Figure 4.6: An alternative fact type with quality annotations. Location participates in an alternative role: that is, for a given person, several possible locations can be recorded.

Table 4.7: Example instantiation of the fact type shown in Figure 4.6.

<i>Person</i>	<i>Location</i>
Michelle Williams	78-123
Michelle Williams	78-122
Emma May	98-122
Emma May	98-123
Emma May	45-234
Jane Bennet	78-222

to the same person. Therefore, there are three alternative sets in this instance: a two-member set consisting of the facts shown in the first two rows, a three-member set consisting of the following three facts, and a singleton set consisting of the last fact.

In some cases, it is desirable to combine alternative and temporal fact types. In this instance, the alternative uniqueness constraint may be either a lifetime or snapshot constraint. The first case is straightforward, but the second has slightly special semantics, as time comes into play when determining whether given facts are alternatives. Two facts f_1 and f_2 belonging to an alternative fact type that has a snapshot alternative uniqueness constraint are regarded as alternatives at a given time t if and only if:

- $st_1 \leq t \leq et_1$ and $st_2 \leq t \leq et_2$ where $[st_1, et_1]$ and $[st_2, et_2]$ are the timestamps associated with f_1 and f_2 , respectively; and
- the objects participating in each of the roles covered by the snapshot alternative uniqueness constraint are identical.

4.4.5 Fact dependencies

Dependencies have been widely studied in database theory, particularly in conjunction with the relational model. They are concerned with capturing constraints on the values of attributes (columns) in relations⁵. A common example is the functional dependency, which has been mentioned in passing in the preceding sections. Formally, a functional dependency, written $X \rightarrow Y$, expresses a constraint that for a given set of attributes X and a set of attributes Y , the values of X uniquely determine those of Y . The enforcement of such dependencies helps to ensure the integrity of information stored in a database.

The type of context dependency described in this section differs from standard database dependencies in at least two important ways. First, fact dependencies capture relationships between entire facts rather than between sets of attributes/roles. Second, they do not express constraints that can be rigidly enforced in the manner of traditional database dependencies. Rather, they provide hints that can assist a context management system in preserving the consistency of a context instance in cases in which the enforcement of a stronger constraint is not possible.

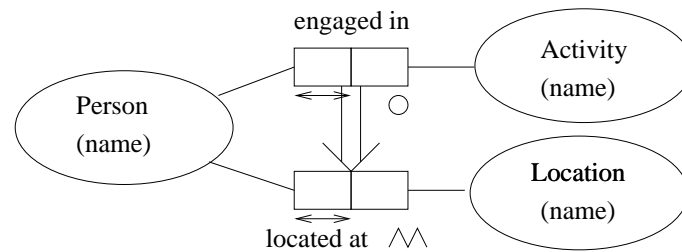
Informally, a dependency, f_2 *dependsOn* f_1 , between a pair of distinct *dynamic, non-derived* facts indicates that the state of the first fact, f_2 , is partially determined by that of the second, f_1 . Generally, changes in the state of f_1 will be followed by corresponding changes in the state of f_2 ⁶. A state change consists of either:

- the insertion of a new fact;
- the modification one or more roles of a fact; or
- the deletion of a fact.

Fact dependencies are modelled when the nature of the relationship between the facts is imprecisely known; otherwise, f_2 could simply be defined as a derived fact. Knowledge of such fuzzily-defined relationships is made relevant by the nature of context-aware pervasive systems, which, unlike the domains traditionally considered by the database community, involve large volumes of frequently changing information, a large subset of which is not precisely known, but instead estimated using sensors. In such systems, it is impossible to expect perfect consistency of context information: a more reasonable goal is to manage changes to promote the greatest possible consistency. Dependencies can be used by a context management system to respond to a change in the state of a fact that has one or more dependents as follows:

⁵Note that attributes in the relational model closely correspond to roles in ORM/CML.

⁶Note that such changes will not always be observed, for example, when the change in f_1 is too small to have a noticeable effect on f_2 .



$\text{engaged in}(p1,a) \text{ dependsOn } \text{located at}(p2,l) \text{ iff } p1 = p2$

Figure 4.7: Fact dependency example. A person's activity is partially determined by his/her location. A change in location implies that activity may also change.

- *To force an update of a dependent fact.* This may be achieved, for example, by invoking an appropriate sensor to refresh the value of the fact. If updated information is not available, the context management system may instead retain the fact in its previous state with appropriately degraded quality indicators.
- *To notify interested context consumers of a change in a dependent fact.* If the context management system can determine the new state of the dependent fact (and this is different to the previous state), the new information can be disseminated to consumers. Otherwise, the consumers can be notified of a probable change in the fact, and can choose to respond accordingly.

Note that, as the purpose of dependencies is to manage the change of related fact types, there is no benefit derived from capturing dependencies that involve static fact types, which, of course, do not change.

Dependencies are expressed in CML by specifying:

- the two fact types involved in the dependency. These are indicated by drawing a double arrow from the fact type that participates in the dependent role to the fact type upon which it depends; and
- a logical expression that indicates which precise instances of the two fact types are related by the *dependsOn* relation.

This notation is illustrated in the example shown in Figure 4.7.

4.4.6 Summary

A summary of the modelling notation introduced in Sections 4.4.1 to 4.4.5 is presented in Figure 4.8.

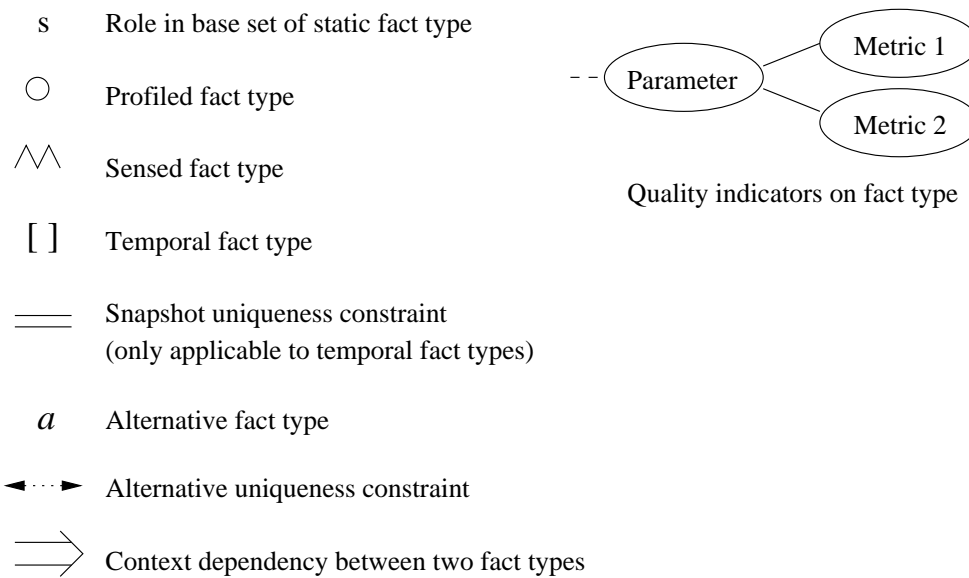


Figure 4.8: Summary of notation.

4.5 Context modelling process

The previous sections presented the extensions that CML adds to FORM in order to capture pertinent characteristics of context information. This section turns to the process associated with constructing a conceptual model using CML. Specifically, it highlights modifications of FORM's CSDP that are required in order to accommodate CML's context modelling constructs.

The original CSDP was presented in Section 4.3.2. This can be adapted for use with CML by refining the steps as follows:

- *Step 2.1.* Draw the corresponding fact types and apply a population check to ensure that fact types are compatible with domain elements.
- *Step 2.2.* For each fact type, analyse the persistence characteristics of the corresponding facts. If the facts are invariant over the lifetime of at least one of the participating object types, mark the fact type as static. Identify the object roles to which the existence of facts is linked; these collectively form the base set, and should be marked with an 's'.

For non-static fact types, determine the most appropriate source for the corresponding facts and annotate them accordingly as derived, sensed or profiled. Any information that can be computed from other facts captured in the context model must be captured by a derived fact type that is accompanied by a derivation rule. The complexity of the derivation rule will help to determine whether the fact type should be explicitly stored (denoted by '**') or computed on demand (denoted by '*'). Frequently changing information, such as

a person's location, is best obtained from sensors, whereas slowly changing information, such as a person's office number, can be profiled.

- *Step 2.3.* For each dynamic fact type, identify sources of possible error and, based on these, annotate the fact type with a set of appropriate quality parameters. For each quality parameter, identify one or more qualitative or quantitative metrics that can be used to evaluate the quality of a fact type with respect to the parameter. The metrics should be chosen to provide a balance between the demands of context consumers for concise and accurate quality measures and the burden on context producers to supply quality information. In general, indirect indicators, such as timestamps to indicate freshness or standard deviations of sensors supplying sensed information, are more realistic metrics than direct quality measures such as probabilistic estimates of correctness.
- *Step 2.4.* For each dynamic fact type, determine the likelihood of conflicting facts. Such conflicts are particularly common among facts derived from multiple sensors (such as location information obtained from a variety of sensor types). If conflicts represent the norm rather than the exception, and there is no satisfactory means of conflict resolution, mark the fact type with the alternative annotation.
- *Step 2.5.* For each dynamic fact type, consider whether it is necessary or desirable to maintain historical information. If so, mark the fact type as a temporal fact type.
- *Step 4.1.* For each temporal fact type, identify relevant snapshot and lifetime uniqueness constraints. For each non-temporal fact type, draw the ordinary uniqueness constraints to capture functional dependencies. Apply an arity check, and split fact types whenever this can be done without loss of information. In general, a fact type should be split if it has a uniqueness constraint that spans fewer than $n - 1$ roles, where n is the arity of the fact type.
- *Step 4.2.* For each alternative fact type, identify one of the uniqueness constraints as the alternative uniqueness constraint, and mark it as such; this constraint must span exactly $n - 1$ roles, where n is the arity of the fact type.
- *Step 4.3.* Identify any fact dependencies that exist between dynamic, non-derived fact types. These occur when the state of one fact type is partially determined by that of another. For each dependency, indicate the pair of fact types that participate by a double arrow between the two types originating from the dependent fact type. Qualify the dependency with an expression that describes which particular fact instances are involved in the dependency.

4.6 Relational mapping

This section now reformulates CML's novel context modelling concepts at the logical level within the framework of the relational model. This brings the modelling constructs closer to an implementation based on a relational database system, and also serves to further formalise the constructs. Section 4.6.1 lays the foundations by presenting a brief formal introduction to the relational model, while Section 4.6.2 describes the basic mapping from FORM to the relational model developed by Halpin. Sections 4.6.3 to 4.6.7 discuss the incorporation of the context modelling concepts into the formal framework of the relational model, extending the mapping procedure presented in Section 4.6.2. The extended mapping procedure is designed with both simplicity and efficient querying in mind. Section 4.6.8 summarises the relational model of context, bringing together the components described in Sections 4.6.3 to 4.6.7. Finally, Section 4.6.9 discusses a possible interpretation of the extended relational model and some of the representational implications that arise from this interpretation.

4.6.1 Overview of the relational model

The relational model, described by Codd in his seminal paper of 1970 [165], is a data model based on the mathematical concept of the relation. Upon this data model a rich theory has been constructed, concerned largely with query and update capabilities and various constraint types. This section reviews a small but core subset of relational theory, restricted to the basic data model and the most fundamental of the integrity constraints. A more detailed discussion of the relational theory extends far beyond the scope of this thesis. Fortunately, this basic core provides a sufficient platform on which to build a relational model of context.

Formal foundations of the relational data model

There are numerous alternative definitions of the relational model, several of which are described by Abiteboul et al. [166]. The form adopted here is roughly aligned with the conventional, named-attribute approach.

The formalisation assumes three disjoint and possibly infinite sets:

- **att**, the set of possible attribute names
- **dom**, the set of possible constant values
- **relname**, the set of possible relation names

A relation schema is simply defined as a relation name belonging to **relname**. Each relation schema corresponds to an ordered list of attributes, denoted $sort(R)$;

the notation $R(a_1, \dots, a_n)$ offers a shorthand that indicates that a relation schema $R \in \mathbf{relname}$ has $sort(R)$ equal to a_1, \dots, a_n , where $a_i \in \mathbf{att}$ for $1 \leq i \leq n$. Here, n is referred to as the arity of R , denoted $arity(R)$. A database schema (also referred to here as a context schema) is a finite set, $C = \{R_1, \dots, R_m\}$, of relation names.

A tuple t over a schema R takes the form of an ordered list $\langle v_1, \dots, v_n \rangle$, such that $n = arity(R)$ and, $v_i \in \mathbf{dom}$ for $1 \leq i \leq n$.⁷ As a shorthand, if the attributes of R are a_1, \dots, a_n , then $t[a_i]$ denotes the value v_i for $1 \leq i \leq n$. Similarly, $t[b_1, \dots, b_j] = c_1, \dots, c_j$ if and only if $\{b_1, \dots, b_j\} \subseteq \{a_1, \dots, a_n\}$ and $t[b_i] = c_i$ for all $1 \leq i \leq j$.

A relation instance r over a schema R can be defined as a finite (and possibly empty) set of tuples over R . A database (or context) instance over a database schema C is a mapping I with domain C , such that $I(R)$ is a relation instance over R for each $R \in C$.

Constraints on the relational model

A database schema can be augmented with constraints that restrict the set of valid instances of that schema. Many classes of constraint have been explored in conjunction with the relational model, some of which are described in [166] and [167]. Only three of the most common are considered here, namely key, entity integrity and inclusion dependency constraints.

Key constraints represent a special class of the functional dependency defined in Section 4.4.5, and are closely related to FORM's uniqueness constraints. A superkey of a relation is a set of attributes whose values, when considered collectively, are unique to a single tuple in a given instance of that relation. Formally, if $R(a_1, \dots, a_n)$ is a relation and the attributes sk_1, \dots, sk_j (where $sk_i \in \{a_1, \dots, a_n\}$ for each $1 \leq i \leq j$) form a superkey of R , then, for any two distinct tuples t_1 and t_2 in an instance r , the following holds:

$$t_1[sk_1, \dots, sk_n] \neq t_2[sk_1, \dots, sk_n]$$

The attributes sk_1, \dots, sk_n are termed a key of R if the result of removing any single attribute, sk_i , $1 \leq i \leq n$, is no longer a superkey of R .

When defining a relation $R(a_1, \dots, a_n)$, all keys of the relation should be specified; this is done by placing a line beneath the relevant attributes. If a relation has multiple keys, one of these keys is explicitly designated the primary key by a double line; this key is used by the database management system to uniquely identify tuples within the relation. Otherwise, if a relation has only one key, this key is implicitly understood to be the primary key.

⁷In practice, tuples are further constrained such that, for a relation $R(a_1, \dots, a_n)$, each v_i ($1 \leq i \leq n$) belongs to a restricted domain, $dom(a_i)$ that is a subset of \mathbf{dom} . Here, the universal domain \mathbf{dom} is used for all attributes for the sake of simplicity, without loss of generality.

Entity integrity constraints are related to primary keys; simply stated, these dictate that no attribute belonging to a primary key may be unknown. The relational model generally captures unknowns using the *null* value, which is a special constant belonging to **dom**⁸.

Finally, an inclusion dependency captures a constraint over two sets of attributes, $\{a_1, \dots, a_n\}$ and $\{b_1, \dots, b_n\}$, where the former set is a subset of the attributes of a relation R_1 and the latter is a subset of the attributes of a relation R_2 . This is sometimes also referred to as a subset constraint. Formally, an inclusion dependency is a constraint of the form $R_1[a_1, \dots, a_n] \subseteq R_2[b_1, \dots, b_n]$ that implies that for each tuple t_1 in $I(R_1)$ there exists a corresponding tuple t_2 in $I(R_2)$ such that the values of t_1 the attributes a_1, \dots, a_n are identical to those of t_2 over b_1, \dots, b_n (that is, $t_1[a_1, \dots, a_n] = t_2[b_1, \dots, b_n]$). Diagrammatically, the constraint is indicated by a dashed arrow originating at the attributes a_1, \dots, a_n and terminating at the attributes b_1, \dots, b_n .

It should be noted at this point that the relational theory has been widely studied for several decades, with the result that variants of the relational model (encompassing a wide variety of constraint types) abound, ranging from the simple to the complex and highly expressive. Within this thesis, the minimal form of the relational model that has just been outlined is always assumed, both because a consideration of complex variants is outside the scope of the thesis, and because many of these variants have little or no support in current database management systems.

4.6.2 Mapping approach

Various mappings from ORM to the relational model have been defined, including Halpin's Rmap procedure [151] which transforms a subset of FORM's modelling constructs to corresponding relational concepts. This section presents a simplified variant of Rmap that does not cover FORM's nested fact types, composite reference schemes or advanced constraint types, but does ensure that the fact abstraction is preserved by maintaining a one-to-one correspondence between fact types and relations⁹.

The mapping is defined as follows. Each fact type, f , is mapped to a relation schema, $R(a_1, \dots, a_n)$, such that n is the number of roles in f and there is a one-to-one correspondence between the attributes a_1, \dots, a_n and the roles in f . R is generally

⁸Note that *null* values need not only designate unknowns: they are commonly also used to indicate that an attribute is not applicable for a given tuple, or that an attribute value is known but not yet recorded [167].

⁹In contrast, Rmap strives to minimise the number of relations in order to reduce the need for expensive joins on relations during querying. The logic-based query approach presented in the following chapter is not based on joins, so the goal of minimising relations is spurious within the context of this work. Instead, the preservation of the fact abstraction within the relational representation is important in allowing logical expressions about context to be formulated in a straightforward and natural fashion.

named according to the label of the fact type. Similarly, the attributes of R generally adopt the names of the object types that participate in the corresponding roles¹⁰.

Internal uniqueness constraints are represented by keys, and, for each relation, one of the keys is selected as the primary key. This should be chosen with the entity integrity constraint in mind; that is, the primary key should be selected such that it does not include any attribute for which the value may sometimes be unknown.

Mandatory roles are mapped to inclusion dependency constraints as follows. An inclusion dependency $R_1[a_1] \subseteq R_2[a_2]$ exists when:

- an object type O participates in roles r_1 and r_2 ; and
- r_2 is a mandatory role; and
- r_1 is mapped to an attribute a_1 in a relation schema R_1 ; and
- r_2 is mapped to an attribute a_2 in a relation schema R_2 .

Figure 4.9 illustrates the mapping procedure using a simple example. In this figure, the conceptual model in (a) is transformed to the set of relations shown below in (b). From this example, it is straightforward to observe the close correspondence between fact types and relations, and, similarly, between uniqueness and key constraints. In the case of the *HasUser* relation, the key constraint spanning the first attribute has been arbitrarily selected as the primary key. The three mandatory role constraints attached to the *Communication Agent* object have been mapped to three equivalence constraints in the relational model; each of these is identical to a pair of symmetric subset constraints. The effect of these three constraints is to ensure that, each time a given communication agent appears in one of the relations, it also appears in the remaining two relations. Note that the external uniqueness constraint shown in (a) between the *has process number* fact type and the *executes on* fact type has no analogue in the relational model shown in (b). In practice, it could be represented by a type of functional dependency; however, it is outside the scope of this thesis to cover constraints not specifically related to the context modelling problem. A discussion of some of the advanced mapping issues associated with FORM can be found in [149]. In the following sections, the basic mapping procedure presented here will be extended to cover CML's novel context modelling constructs.

4.6.3 Classification of fact types

The mapping of static, sensed and profiled fact types to relations is performed according to the standard procedure outlined in the preceding section. The classifica-

¹⁰However, note that a different naming scheme is required when an object type participates more than once in a single fact type. In this case, the attribute names should reflect the different roles played by the object type.

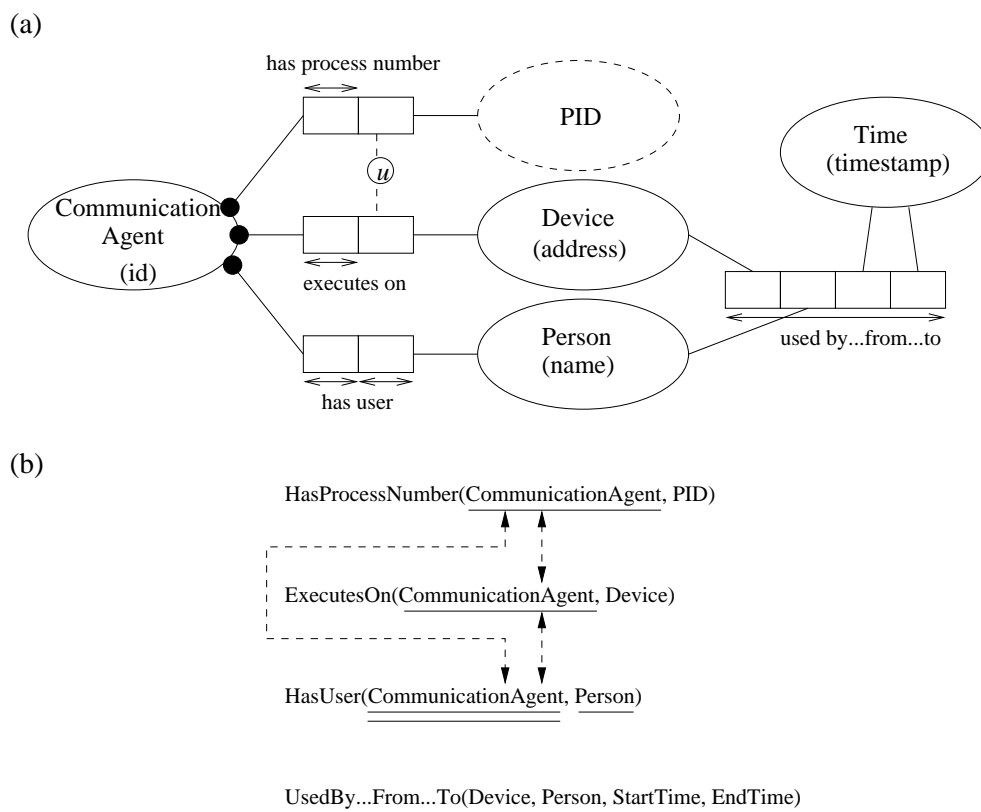


Figure 4.9: (a) A FORM conceptual model, reproduced from Figure 4.2. (b) A relational representation of the model shown in (a).

tion of each fact type represents a form of metadata that is required by the management tasks described in Section 4.4.1 related to update management and resolution of inconsistencies. This metadata can be captured by a total function, *class*, mapping the set of relation schemas, $C = \{R_1, \dots, R_n\}$, to the set $\{static, sensed, profiled\}$.

Derived fact types demand special treatment. In order to define the relational representation of derived facts, it is necessary to introduce a distinction between base and derived relations. These can trivially be defined as relations that represent base and derived fact types, respectively.

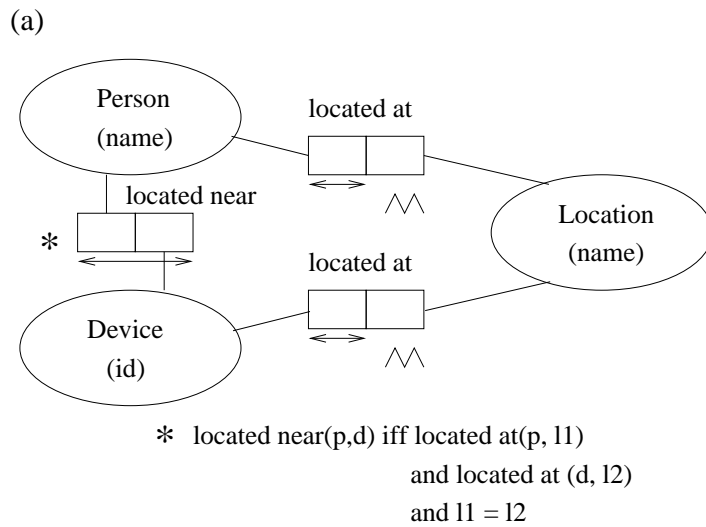
As described in Section 4.4.1, there are two approaches to representing derived fact types. The first approach explicitly stores derived context information, enabling efficient retrieval, but requiring care in order to ensure that consistency is maintained as the base facts change. The second approach involves computing derived facts on demand, thus eliminating the problems associated with consistency, but potentially leading to long query times. Both approaches can be represented in the relational model by views, which can be either materialised or virtual. Materialised views explicitly represent derived information, whereas virtual views adopt the lazy approach of computing derived data on demand.

Views, both materialised and virtual, are generally expressed as queries over the base relations. There are numerous query languages that can be employed for view definition; here, SQL is chosen because of its widespread adoption and its ability to express relatively advanced derivation algorithms (for example, using a range of mathematical aggregation and user-defined functions). Each derived fact type f , having n roles, is mapped to a view represented by a relation schema of the form $V(a_1, \dots, a_n)$ such that there is a one-to-one correspondence between a_1, \dots, a_n and the roles in f , and $a_i \in \mathbf{att}$ for $1 \leq i \leq n$. The derivation rule associated with f is mapped to an SQL SELECT statement, denoted $def(V)$. An example mapping is illustrated in Figure 4.10. Note that derived relations have no associated constraints in the relational representation, as these are implied by the derivation expression and the constraints on the base relations.

4.6.4 Temporal fact types

The mapping of temporal fact types to a relational representation has two main considerations: the need to incorporate timestamps to capture the temporal dimension of the information, and the special treatment required for snapshot uniqueness constraints. These issues are closely related.

Implicit in each temporal fact type is a pair, $(StartTime, EndTime)$, of timestamps corresponding to the endpoints of an interval capturing the valid time of a fact. In the relational representation, this pair becomes explicit. That is, the relation representing a given temporal fact type is formed in the usual way, except



(b)

PersonLocatedAt(Person, Location)

DeviceLocatedAt(Device, Location)

LocatedNear(Person, Device)

def(LocatedNear) =

SELECT PersonLocatedAt.Person, DeviceLocatedAt.Device

FROM PersonLocatedAt, DeviceLocatedAt

WHERE PersonLocatedAt.Location = DeviceLocatedAt.Location

Figure 4.10: (a) Example of a derived fact type, reproduced from Figure 4.3 (d).
(b) A relational representation of the fact types shown in (a).

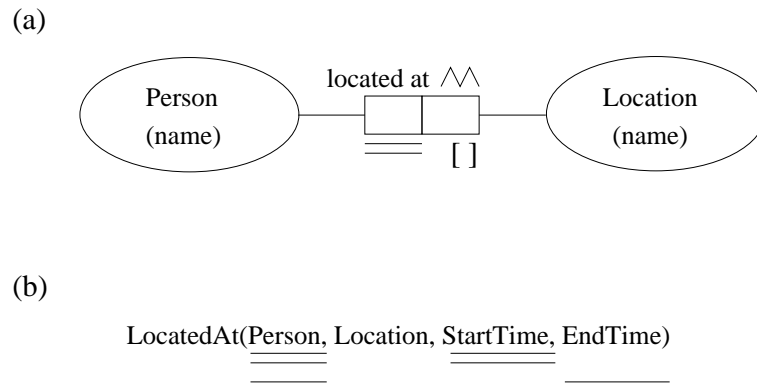


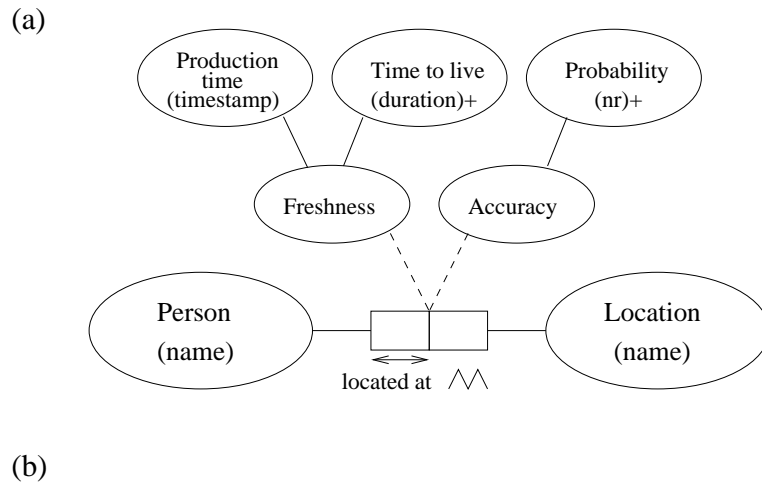
Figure 4.11: (a) Example of a temporal fact type, reproduced from Figure 4.4. (b) A relational representation of the fact type shown in (a).

that two additional timestamp attributes are appended to the relation, as shown in Figure 4.11.

While lifetime uniqueness constraints (as well as other lifetime constraints, such as mandatory role constraints) are mapped as usual, the mapping of snapshot uniqueness constraints represents a special case. A snapshot uniqueness constraint indicates that the combination of objects participating in its roles is unique at any arbitrarily chosen point in time. This constraint cannot be fully captured in the basic relational model that was outlined in Section 4.6.1; however, it can be captured in part by a pair of key constraints that cover the roles spanned by the snapshot uniqueness constraint combined with the *StartTime* and *EndTime* attributes, respectively. An example mapping of a snapshot uniqueness constraint appears in Figure 4.11; in this example, the key spanning the *Person* and *StartTime* attributes has been selected as the primary key. Note that the two key constraints only enforce uniqueness with respect to the endpoints of the valid time interval, and do not cover the open interval (*StartTime*, *EndTime*) falling in between. The enforcement of the constraint over all of these points in time would be both difficult to express and computationally expensive to enforce in most database management systems.

4.6.5 Quality

The quality metrics for a given fact type characterise the types of concrete quality value recorded for each fact belonging to that type, and are partitioned amongst one or more quality parameters. The mapping of a quality-annotated fact type to a relational representation can be performed in a straightforward fashion. This mapping treats quality information identically to other types of context information, rather than as special metadata, enabling uniform querying and updating of both types of information. Each fact type is mapped to a relation as usual, and then augmented with an additional attribute for each quality metric. This is illustrated



LocatedAt(Person, Location, ProductionTime, TimeToLive, Probability)

Figure 4.12: (a) A fact type with quality annotations, reproduced from Figure 4.5. (b) A relational representation of the fact type shown in (a).

in Figure 4.12 using the example introduced in Section 4.4.3. Observe that the quality metrics are mapped in essentially the same manner as objects participating in ordinary roles of the fact type. The quality parameters are generally ignored unless two or more parameters have identically named metrics, in which case the attribute name for each metric is formed by prepending the parameter name onto the metric name. Indeed, it is not strictly necessary to include quality parameters even at the conceptual level; however, doing so affords additional clarity, and enables the grouping of related quality metrics.

4.6.6 Alternatives

The mapping of alternative fact types is non-trivial, as their semantics are at odds with the usual semantics of the relational model. Ordinarily, relations capture only those facts that are assumed to be truthful; in contrast, the alternative fact type captures a set of mutually exclusive facts. That is, of a set of n alternative facts, at most one is truthful, and at least $n - 1$ are false. In order to capture alternative fact types, it is necessary to either substantially modify the relational model, or apply a somewhat different interpretation to the facts captured by the standard relational model. The latter approach is adopted here, as it enables much of the relational database theory to be retained without modification and is largely compatible with existing relational database implementations. This section describes the representation of alternative facts within the framework of the relational model, while Section 4.6.9 covers issues related to semantics and interpretation.

The mapping of alternative fact types is performed as usual, with the exception

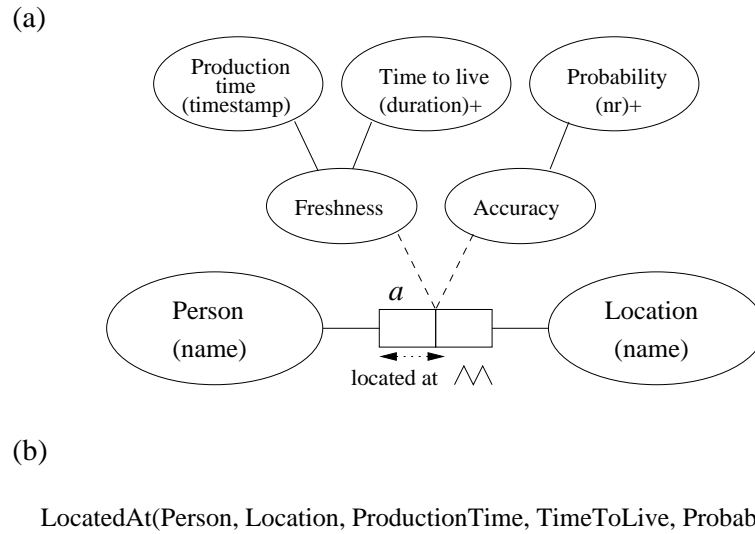


Figure 4.13: (a) An alternative fact type, reproduced from Figure 4.6. (b) A relational representation of the fact type shown in (a).

that special treatment is required in relation to the alternative uniqueness constraint. In general, internal uniqueness constraints are mapped trivially to key constraints in the relational representation. This straightforward mapping is inadequate in this case. Instead, a key is formed from the alternative uniqueness constraint by extending this constraint to cover the alternative role, as shown in Figure 4.13.

The alternative role takes the form of metadata that enables the identification of alternative facts. Letting A be the set of alternative relations, $altRole$ is a total function mapping A to **att**, that, for each relation R in A , yields the alternative role of R .

Special care is required when combining alternative and temporal fact types. There are two possibilities. The first is that the alternative uniqueness constraint takes the form of a lifetime constraint, in which case the mapping is performed as usual. In the second case, in which the alternative uniqueness constraint takes the form of a snapshot constraint, the two keys formed from the uniqueness constraint encompass the following roles:

- the roles covered by the uniqueness constraint combined with the alternative role and the *StartTime* timestamp; and
- the roles covered by the uniqueness constraint combined with the alternative role and the *EndTime* timestamp.

4.6.7 Fact dependencies

Fact dependencies represent a form of metadata or loose constraint about context information that is primarily important for context management purposes, as de-

scribed in Section 4.4.5; however, there are circumstances in which applications can benefit from a knowledge of fact dependencies when making context-aware choices, so the representation of dependencies that is adopted here is chosen to support both uses. One of the principal uses of fact dependencies for applications is to enable the flow-on effects of a particular course of action to be anticipated, leading to more informed choices. For example, when adapting bandwidth usage patterns, the knowledge that battery lifetime will be affected can be taken into consideration, and can help to prevent stability problems such as the cascading adaptations described by Efstratiou et al. [75].

Like ordinary context information, dependency types are fixed while their instances vary over time. As described in Section 4.4.5, a dependency type can be described by a logical expression on a pair of fact types. The instance of a fact dependency is a binary relation on facts of the two types (that is, a set of fact pairs). A dependency instance changes as the base facts change, and hence should not be explicitly stored. Instead, the approach taken here is to represent fact dependencies in a similar manner to derived fact types. This representation employs the concept of unique fact identifiers, thus enabling a uniform representation of dependencies regardless of the fact types involved, as well as straightforward identifier-based querying. Additionally, the use of fact identifiers has efficiency benefits, as it allows facts to be rapidly located and retrieved using indexing mechanisms; this is important as pervasive computing environments can contain large numbers of frequently changing facts and large numbers of dependencies, which need to be managed in a timely fashion.

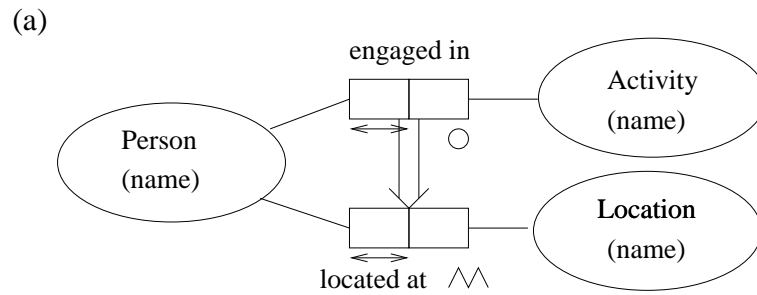
The following assumptions are made about fact identifiers. Each fact is assigned an identifier belonging to the set FID which is a subset of \mathbf{dom} . For a fact f , this identifier is denoted $id(f)$. Each identifier is unique to a single fact in a context instance; expressed more formally, if f_1 is a fact belonging to relation R_1 with primary key pk_1 and f_2 is a fact belonging to relation R_2 with primary key pk_2 , then the following constraint holds over all context instances:

$$id(f_1) = id(f_2) \text{ implies } R_1 = R_2 \text{ and } R_1[sk_1] = R_2[sk_2]$$

To preserve this constraint, each fact is assigned a previously unused identifier upon insertion into a context instance. This is retained until deletion, or until one or more of the primary key attributes is modified. The latter case can be viewed as a deletion followed by an insertion, with the newly inserted fact being assigned its own new identifier.

In theory, the dependencies in a context instance can be completely characterised by a single, binary *dependsOn* relation defined as a view using an appropriately constructed derivation rule¹¹. The main problem with this approach is that it is

¹¹This approach would effectively merge the dependency definitions captured in the conceptual



engaged in(p1,a) dependsOn located at(p2,l) iff p1 = p2

(b)

```

EngagedIn(Person, Activity)
LocatedAt(Person, Location)
LocatedAtDependents(FactID, dependentID)
def(LocatedAtDependents) =
  SELECT LocatedAt.ID, EngagedIn.ID
  FROM EngagedIn, LocatedAt
  WHERE EngagedIn.Person = LocatedAt.Person

```

Figure 4.14: (a) Example of a fact dependency between two binary fact types, reproduced from Figure 4.7. (b) A relational representation of the two fact types and the fact dependency shown in (a).

inefficient to compute this (potentially very large) relation on demand each time the set of dependents for a given fact are required (or, alternatively, to maintain this relation as a materialised view that would potentially need to be updated upon each insertion and deletion involving a fact type that participates in any dependency). A more efficient approach is to partition the *dependsOn* relation by fact type. This approach is illustrated in Figure 4.14. Each fact type that is the target of a fact dependency has a corresponding binary relation with attributes *factID* and *dependentID*. In the figure, each of the fact types is mapped to a relation as usual, while the dependency is captured by the view, *LocatedAtDependents*. The latter is defined in a similar fashion to the derived fact type presented in Section 4.6.3, using SQL. The SQL selection statement is a straightforward mapping of the logical expression in (a). Note the use of the special *ID* attribute, which is not explicitly listed in the definitions of the *EngagedIn* and *LocatedAt* relations, but is understood by the context management system to represent the system-assigned identifier of the corresponding facts.

The simple example of Figure 4.14 only deals with a single fact dependency; in model into a very large SQL SELECT statement.

Table 4.8: Example instantiation of the *EngagedIn* relation shown in Figure 4.14 (b).

<i>Person</i>	<i>Activity</i>	<i>Fact ID</i>
Michelle Williams	On lunch	3095893
Emma May	Writing thesis	3098599
Jane Bennet	On lunch	3098974
Mark Darcy	In meeting	3098347

Table 4.9: Example instantiation of the *LocatedAt* relation shown in Figure 4.14 (b).

<i>Person</i>	<i>Location</i>	<i>Fact ID</i>
Michelle Williams	Main Refec	3092874
Mark Darcy	78-633	3091737
Emma May	45-234	3098880

Table 4.10: Example instantiation of the *LocatedAtDependents* relation shown in Figure 4.14 (b).

<i>FactID</i>	<i>DependentID</i>
3092874	3095893
3091737	3098347
3098880	3098599

cases involving a fact type that is the target of two or more dependencies, these are merged into a single view using an SQL statement that combines the corresponding dependency expressions.

Example instantiations of the three relations of Figure 4.14 (b) are illustrated in Tables 4.8 to 4.10. The third column in Tables 4.8 and 4.9 shows the fact identifier, which represents a form of metadata that does not appear explicitly within the relations.

4.6.8 Summary

This section summarises the relational representation of context developed in Sections 4.6.2 - 4.6.7.

An abstract context model (or schema) can be characterised as an extended relational model as follows. Assuming the disjoint and possibly infinite sets **att**, **relname** and **depname**, representing the sets of possible attribute names, constant values, relation names and dependency names, respectively, a context model comprises:

- A finite set $C = \{R_1, \dots, R_n\}$ of base relations, with $R_i \in \mathbf{relname}$ for $1 \leq i \leq n$.
- A set of derived relations $DC = \{R_1, \dots, R_p\}$, such that DC is disjoint with C

and $R_i \in \mathbf{relname}$ for $1 \leq i \leq n$.

- A total function, $sort$, mapping each relation in $C \cup DC$ to an ordered list of attributes, a_1, \dots, a_m , such that $a_i \in \mathbf{att}$ for $1 \leq i \leq m$.
- A total function, $arity$, on $C \cup DC$ such that for R in C , $arity(R)$ is m , the number of attributes in $sort(R)$.
- A set K of key constraints. Each key is characterised by a base relation name, $R \in C$, and a set of attribute names that represents a subset of $sort(R)$.
- A set PK of primary key constraints, such that $PK \subseteq K$ and each relation R in C appears exactly once.
- A set ID of inclusion dependency constraints. Each constraint is characterised by a source relation $R_1 \in C$, a set of attributes $\{a_1, \dots, a_j\}$ that is a subset of $sort(R_1)$, a target relation $R_2 \in C$ and a set of attributes $\{b_1, \dots, b_j\}$ that is a subset of $sort(R_2)$.
- A total function, $class$, mapping the set of base relations, C , to the set of special values $\{static, sensed, profiled\}$.
- A total function def on DC that maps each derived relation to an SQL SELECT statement that defines how the relation is computed from the base relations in C .
- A set $A \subseteq (C \cup DC)$ of relations that represent alternative fact types.
- A total function $altRole$ that defines the alternative role for each relation in A such that, for all $R \in A$, $altRole(R) \in sort(R)$.
- A set $FD = \{D_1, \dots, D_q\}$ of fact dependencies, with $D_i \in \mathbf{depname}$ for $1 \leq i \leq q$.
- A total function, $depDef$ on FD that maps each fact dependency to an SQL SELECT statement that defines the ordered pairs of facts (in terms of fact identifiers) that are linked by the $dependsOn$ relation.

Assuming a possibly infinite set of constant values, \mathbf{dom} that is disjoint from $\mathbf{att} \cup \mathbf{relname} \cup \mathbf{depname}$, a context instance can be defined as follows. A tuple, or fact, over a relation schema R is an ordered list of values, $\langle v_1, \dots, v_n \rangle$, such that $n = arity(R)$ and $v_i \in \mathbf{dom}$ for $1 \leq i \leq n$. A relation instance over R is a finite set of tuples over R . For a given context schema, an *extensional* (or base) instance of this schema, I_E , is a total function on the set of base relations, C , such that for each relation R in C , $I_E(R)$ is a relation instance over R and, further,

the relation instances occurring in the range of I_E collectively satisfy the key and inclusion dependency constraints defined by K and ID , respectively.

The complete, or *intensional* instance of a schema, I_I , formed from I_E , is a total function on $C \cup DC \cup FD$ such that:

- if $R \in C$ (R is a base relation), then $I_I(R) = I_E(R)$; else
- if $R \in DC$ (R is a derived relation), then $I_I(R)$ is computed from I_E according to $def(R)$; else
- if $R \in FD$ (R captures a set of fact dependencies), then $I_I(R)$ is computed from I_E according to $depDef(R)$.

Each fact f belonging to a relation of a context instance I_I has a corresponding unique identifier $id(f)$ belonging to the set ID (a subset of **dom**).

4.6.9 Interpretation

In order to exploit and reason about information captured using the extended relational model developed in the preceding sections, assumptions about the semantics of the model must be introduced. Such assumptions can be exploited to form an *interpretation* of a context instance.

It is common practice, when interpreting a relational database as a set of logical facts, to employ a completion rule in order to deduce information that is not explicitly represented; often, this is based on a closed world assumption (CWA) [168]. Expressed simply, this assumption states that if a fact is present in a database, its truth is assumed; otherwise, its negation holds¹². Generally, the CWA applies only to logical queries, termed *assertions*, over known fact types and known domain objects; when this condition is not met, the CWA offers no information. The CWA affords a significant degree of power: given any assertion that meets the aforementioned condition and a database of information, the truth or falsity of the assertion can be determined, without the need to explicitly store negative facts. For this reason, the CWA is widely adopted, including in the following chapter, where it forms one of the premises of the situation abstraction. Unfortunately, the CWA introduces at least two significant problems, both of which arise from the incomplete and imperfect nature of context information. The first is a variant of a well-known problem related to incompleteness, while the other, related to the alternative fact type, is peculiar to the context representation adopted here. These problems, some candidate solutions, and some implications of the solutions on the representation of context, will now be described.

¹²This is similar to the negation as failure rule employed in logic programming.

Incompleteness and the closed world assumption

The simple form of the CWA that has been described is unable to deal with unknown context information. This can be seen easily using an example. Suppose that an assertion about a user's location, $LocatedAt["Emma May", "98-122"]$ ¹³ is true but unknown. If no information about Emma May's location is recorded by a the context instance, then the CWA erroneously implies that $\neg LocatedAt["Emma May", "98-122"]$ holds.

As mentioned previously, this problem is well known, and generally overcome by:

- introducing an explicit representation of unknown information, generally using the special *null* value (for example, $LocatedAt["Emma May", null]$); and
- applying a three-valued logic when evaluating assertions, where the third value denotes *unknown* or *possibly true*. Using this approach, assertions such as $LocatedAt["Emma May", "98-122"]$ and $LocatedAt["Emma May", "98-233"]$, when evaluated against a context instance containing the tuple $\langle "Emma May", null \rangle$, would both yield this special value, rather than the usual values of *true* or *false*.

This approach can be adopted here, with special care given to the representation of temporal fact types. In particular, histories must be complete, in the sense that any gap implies that the fact type is not applicable at the corresponding time (rather than that the actual state is unknown).

Alternatives and the closed world assumption

A second problem that has a less obvious solution is that of how to reconcile the CWA with the alternative fact type. As presented, the CWA regards any fact that is explicitly present in a context instance as truthful. This is at odds with the semantics of the alternative fact type, which instead captures sets of n alternative facts, of which at most one is true. One approach to overcome this discrepancy is to amend the CWA as follows:

- an assertion over a fact belonging to an alternative fact type evaluates to *false*, as usual, provided that no matching fact is present in the context instance;
- otherwise, the assertion evaluates to *true*, as usual, if a matching fact is present in the context instance and this fact has no alternatives;

¹³This effectively has the same semantics as asserting $\langle "Emma May", "98-122" \rangle \in I_I(LocatedAt)$, where I_I is a context instance containing the relation $LocatedAt$. This notation will be discussed further in the following chapter and in Appendix A.

- otherwise, the assertion evaluates to the third logical value, *possibly true*.

This solution brings the CWA closer to the semantics of the alternative fact type, but is not entirely satisfactory. Consider the alternative set from the example of Table 4.7, $\{ \langle \text{“Emma May”, “98-122”} \rangle, \langle \text{“Emma May, 98-123”} \rangle, \langle \text{“Emma May, 45-234”} \rangle \}$. Each of the facts is regarded as *possibly true*, meaning that the CWA would likewise assign a value of *possibly true* to the assertion $LocatedAt[\text{“Emma May”, “98-122”}] \vee LocatedAt[\text{“Emma May”, “98-123”}] \vee LocatedAt[\text{“Emma May”, “45-234”}]$ ¹⁴, rather than the *true* value suggested by the semantics of the alternative fact type¹⁵. One approach to overcome this anomalous result is to abandon the relational representation of context entirely, and instead adopt a representation based on the Horn clauses of deductive databases. The work of Kong and Chen [146], dealing with incomplete constants in relation to definite deductive databases, addresses a problem that is remarkably similar to that considered here in a manner that avoids the aforementioned anomaly; unfortunately, this solution is both significantly more complex and computationally expensive than the alternative described above. As the consideration of such heavyweight solutions falls outside the scope of this thesis, the simpler approach will be adopted, and the investigation of alternative approaches will be left as an area for future research.

4.7 Discussion and future work

This chapter presented a series of novel context modelling constructs, which were formulated as extensions to the FORM conceptual modelling approach developed by Nijssen and Halpin for the design of information systems. The resulting modelling notation and methodology, CML, can be used to produce a rich formal specification of the context requirements of a context-aware application. The context modelling constructs and their associated context management issues were presented in an abstract way in Section 4.4, and then mapped to a relational representation in Section 4.6. Chapter 6 will demonstrate that the relational representation can be implemented in a straightforward fashion as an extended relational database.

CML has numerous benefits over the alternative context modelling techniques surveyed in Section 3.2. First, it is both more formal and expressive, supporting all of the key characteristics of context information introduced in Section 2.2. The four types of context information described in Section 2.2.1 (static, sensed, profiled and derived) are explicitly distinguished using the annotations described in Section 4.4.1. Uncertainty is addressed in three separate ways. Unknown context is explicitly

¹⁴ Assuming the usual semantics of disjunction under a three-valued logic.

¹⁵ A similar problem exists for conjunction; for example, $LocatedAt[\text{“Emma May”, “98-122”}] \wedge LocatedAt[\text{“Emma May”, “98-123”}] \wedge LocatedAt[\text{“Emma May”, “45-234”}]$ also evaluates to *possibly true*, rather than the expected value of *false*.

represented as described in the previous section, information quality is characterised as described in Section 4.4.3, and ambiguous information is expressed as alternative facts as outlined in Section 4.4.4. Historical context information is captured using the temporal fact type introduced in Section 4.4.2, and dependencies amongst context facts are captured using fact dependencies as discussed in Section 4.4.5.

A further advantage of CML over the modelling solutions described in Section 3.2 is that it is capable of supporting a variety of tasks across the software engineering lifecycle. The graphical modelling notation is well suited for use in analysis and design tasks (as shown in the case study presented in Chapter 7), the relational representation to run-time context management tasks, and the situation abstraction (introduced in the following chapter) as a tool for describing and programming with context in high level terms.

Finally, by aligning the context modelling approach with established information modelling techniques, there is considerable scope for leveraging information systems solutions for dealing with uncertainty, information quality, temporal data, and rapidly changing sensor-derived data, as described in Section 4.2.

Although CML has been substantially refined and improved since it was first published in [40], there remain numerous areas for future work. First, some of the problems associated with the interpretation of uncertainty in the relational representation in general, and the alternative construct in particular, were outlined in Section 4.6.9. A rudimentary solution was presented, but further work is needed to evaluate alternative solutions. Additionally, scalability is an issue that has not been addressed, yet is crucial in pervasive systems. Like FORM, CML assumes that developers model a relatively small set of context requirements in relation to a single application. The integration of such models to build large-scale context-aware systems is an open and challenging research problem. Finally, there is an urgent need to consider the privacy issues inherent in context-aware systems. Context and privacy models need to be interwoven to prevent abuses of context information. A suitable privacy model for context-aware systems should cover not only access control, but should also limit information gathering practices, including the use of sensors to monitor people and environments, and govern the storage and use of context information once it is acquired by consumers. There have been several recent efforts to develop privacy models for pervasive computing environments in general [134, 169–171]; however, privacy support in context-aware systems is generally either non-existent or restricted to access control mechanisms [24, 105, 172–176]. Robinson and Beigl [177] propose a more sophisticated security architecture founded on the notion of trust; however, this is not yet fully implemented and has several known weaknesses. Research on privacy in databases is currently in its early stages [178], and is likely to address a subset of the privacy requirements of context management systems.

Chapter 5

Programming abstractions

The context modelling approach described in the previous chapter allows contexts to be described in a precise and fine-grained fashion. The fact abstraction that forms the basis for this approach is well suited for use in context management tasks; however, it is less appropriate for use as a programming abstraction, as individual facts are generally far removed from people's high-level conceptualisations of context. This chapter is concerned with the development of a model that allows contexts to be described in more abstract and general terms, and also with the identification of complementary programming techniques, in order to simplify the task of the application developer in constructing context-aware applications that are flexible and easily customised.

The structure of the chapter is as follows. Section 5.1 describes current problems with the development of context-aware software, and motivates the need for appropriate programming abstractions and models. In addition, it presents a brief overview of the solutions proposed in this chapter. Section 5.2 describes the situation abstraction, which allows abstract contexts to be specified in terms of CML's facts. In Section 5.3, two approaches to programming with this abstraction are outlined. The first model, based on triggering, has been widely explored previously in connection with adaptive and context-aware software, as discussed in Chapters 1 and 3. Section 5.3.1 demonstrates that the situation abstraction easily supports this widely used programming model and offers an original solution for dealing with uncertain context information. Section 5.3.2 introduces a more novel programming model, based on context-dependent branching, and motivates the exploitation of user preference information in conjunction with this model in order to derive the flexibility and autonomy required of pervasive computing software. In Section 5.4, the requirements of a preference model for context-aware systems are laid out, and a model that satisfies these is developed. Finally, Sections 5.5 and 5.6 conclude the chapter by summarising the contributions of the work described in Sections 5.2 to 5.4 and outlining topics for future work.

5.1 Motivation and approach

The crucial role of appropriate abstractions and programming models in the success and adoption of context-aware applications has been long recognised. For example, Brown et al. argued in 1997 that “if the market opportunity for context-aware applications is to be taken, it must be made easier to create applications” [28]. This view has been reiterated several times since [31, 40, 111] and is backed by a wider recognition of the inadequacy of current programming techniques for pervasive computing software in general [66, 179–182]. However, as the survey in Section 3.3 illustrated, very little progress has been made in this area. While significant attention has been devoted to the development of context gathering infrastructures, the programming models used in conjunction with these remain inadequate and often oversimplified.

At the same time, there is a growing awareness of common usability problems associated with context-aware software [115–117, 160, 183, 184]. These pose significant design challenges. Perhaps the largest challenge lies in exploiting context information such that applications are highly autonomous, requiring little user input, while at the same time ensuring that control ultimately lies with the user. In order to achieve this balance, there is a need for transparency, such that the actions of applications are visible to users and appear consistent and predictable [183], as well as for feedback mechanisms that allow users to override any inappropriate actions. Personalisation is also a key consideration in ensuring that context-aware applications are acceptable to users [185]; that is, application developers must recognise that users have differing requirements, and that these may evolve over time.

These software engineering and usability challenges provide the motivation for the research presented in this chapter. The first part of the chapter is concerned with the development of a high-level model of context that forms a suitable tool with which to describe and program context-aware behaviour. This model is founded on the assumption that the fact-based model of the previous chapter is not the right abstraction for this task, and that a more abstract model, which is closer to the terms in which people conceptualise context, is more appropriate. The key goals of the model are the following:

- to support the description of context in terms of abstract classes that are delineated according to the selected aspects (fact types) that are relevant to the tasks carried out by the context-aware application and/or user; and
- to allow classes of context to be easily combined, such that complex contexts can be described with minimal effort.

The situation abstraction described in Section 5.2 satisfies these goals and is fully compatible with the fact-based model of the previous chapter.

The situation abstraction provides a useful way of describing contexts at a level of detail suited the needs of users and application developers, but by itself does not provide an adequate means to describe context-aware behaviour. This latter task is addressed in Section 5.3. Here, two complementary styles of programming that can be used in conjunction with the abstraction are explored. The first (the triggering model) employs an event-based paradigm in which actions are associated with relevant context changes. The second (the branching model) allows decision points to be placed in application logic such that different paths or branches are executed depending on the context.

As the triggering model has been widely researched [23,28–35], this model is not explored here in great depth. Section 5.3.1 instead demonstrates that the situation abstraction introduced in Section 5.2 is compatible with this widely used programming model, and enables a novel form of support to be provided for imperfect context information. In contrast, the branching model has received almost no recognition as an important programming model for context-aware systems. Consequently, it receives a more thorough treatment in Sections 5.3.2 and 5.4. In Section 5.3.2 it is argued that, in order to support the autonomy and flexibility required in pervasive computing applications, branching decisions should account for user preferences, and these should be specified in a flexible manner so that they can be customised and evolved over time. Section 5.4 addresses the development of a model of user preferences suited to the requirements of the branching problem. Like the situation abstraction, this model is designed to be conceptually simple and to support reuse based on composition.

5.2 The situation abstraction

5.2.1 Overview

Chapter 3 demonstrated that most context-aware systems are built upon software infrastructures that describe contexts using informal, fine-grained, and often piecemeal modelling approaches, and frequently require applications to perform multiple explicit queries in order to obtain rich types of context information. This is despite the fact that, in general, higher level abstractions are considerably easier for developers to use [111]. The fact-based modelling approach of the previous chapter partially addresses this problem; however, it remains inadequate as a programming abstraction, as discussed in previous sections. The situation abstraction fills this gap. It allows abstract classes of context to be easily defined in order to capture conditions or scenarios that are interesting to application developers and users. The abstraction is layered on top of CML, such that situations are expressed as predicates formulated in terms of atomic facts. A situation is said to hold in a given

context exactly when the situation predicate is satisfied (that is, made true) by the fact-based description of the context.

Situations are defined using the computationally complete and decidable variant of predicate calculus that is outlined in the following section. Situations can be combined using the logical operators (conjunction, disjunction and negation) to form richer situations. This feature allows for reuse and enables users to construct complex context descriptions in a relatively straightforward manner. A variety of issues involved in combining situations form the subject of Section 5.2.4.

During the development of CML in the previous chapter, the key characteristics of context information introduced in Section 2.2 were important considerations. These also motivate the design of the situation abstraction. In particular, special treatment is required for imperfect context information, as discussed in Section 5.2.5.

The situation abstraction is not the only context modelling approach founded upon predicate calculus, as seen in Chapter 3. Similar modelling approaches have been explored by Dey et al. in relation to their context-aware reminder application [16,111] and by Ranganathan et al. in the scope of a context-aware chat tool [10]. These approaches were introduced in Section 3.3, but are further analysed in Section 5.2.6 and compared with the situation abstraction presented in Section 5.2 in terms of efficiency and expressiveness.

5.2.2 Defining situations

The situation abstraction is used to specify classes of context by placing constraints on relevant parameters, such as the time of day, location or activity of the user. These abstract contexts, termed situations, are expressed as predicates on the state of the context using a novel variant of predicate calculus. The syntax and semantics of the modified calculus are defined fully in Appendix A, but are also summarised here. This approach to describing abstract contexts was chosen with the aim of balancing simplicity and ease of use with expressive power. Only a small set of core logical and arithmetic operators are directly supported, but others can be introduced as required in the form of user-defined functions.

The simplest situation expressions, referred to collectively as the base situation formulas, for a given context model with relations $\mathbf{R} = C \cup DC \cup FD$ (where C , DC and FD denote the base relations, derived relations and fact dependencies, respectively, as described in Section 4.6.8) are:

- boolean expressions of the form $t_1 = t_2$, $t_1 \neq t_2$, $t_1 < t_2$, $t_1 \leq t_2$, $t_1 > t_2$ or $t_1 \geq t_2$, where t_1 and t_2 are terms; or
- *atoms* over the set of relations, \mathbf{R} . These are expressions of the form $p[t_1, \dots, t_n]$, where $p \in \mathbf{R}$ and each t_i ($1 \leq i \leq n$) is a term or the special anonymous

variable (written $_$). Atoms are essentially a form of assertion or query over a context fact type.

The terms in these base formulas are either constants (belonging to the set **dom**), variables (belonging to the set **var**) or functions. The latter are arithmetic expressions on terms ($t_1 + t_2$, $t_1 - t_2$, $t_1 \times t_2$ or $t_1 \div t_2$) or expressions of the form $f(t_1, \dots, t_n)$, where f is a function name and each t_i is either a term or an ordered list of terms. Each function f is either a built-in (predefined) or user-defined function. The former class includes the *id* function, which yields the fact identifier for the fact that matches the supplied relation name and list of attribute values (provided such a fact exists), *timenow*, which returns the current date and time, and *isnull*, which indicates whether the value of a given variable is null (unknown). User-defined functions can perform arbitrary computations, thereby allowing situations to capture complex conditions that cannot be expressed using the logical and arithmetic operators alone.

The base formulas can be combined recursively using conjunction (\wedge), disjunction (\vee) and negation (\neg). These logical connectives assume their normal properties in terms of idempotency, commutativity, associativity, distributivity, and so on. The following restricted forms of the universal and existential quantifiers are also permitted:

- $\forall x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \varphi$
- $\exists x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \varphi$

where $\{x_1, \dots, x_i\} \subseteq \{t_1, \dots, t_n\}$, φ is a valid situation formula and $p[t_1, \dots, t_n]$ is an atom, as described above¹. The atom serves as a constraint on the values of the variables, and is present for reasons of efficiency and safety which will be discussed in Section 5.2.6.

A situation *predicate* is simply a named formula containing a set of free variables, v_1, \dots, v_n , written as follows:

$$S(v_1, \dots, v_n) : \varphi$$

Here, S is the situation name and φ is a well-formed situation formula in which the set of free variables is exactly $\{v_1, \dots, v_n\}$.

The evaluation of a situation predicate is performed with respect to a set, v , of concrete bindings for the variables, v_1, \dots, v_n , (termed a *valuation*) and an intensional context instance, I_I , according to the assumptions outlined in Section 4.6.9. In most cases, the context instance is the set of information that is available through a context management system, whereas the variable bindings are additional parameters

¹Note that the symbol “•” acts as a separator and has no special semantics here.

specified by the application (application context) or by the user (user context). In the absence of alternative facts and unknown (null) values in the relevant relations of I_I , the evaluation is straightforward. The logical connectives (\vee , \wedge and \neg), arithmetic operators ($+$, $-$, \times and \div) and comparison operators ($=$, \neq , $<$, \leq , $>$ and \geq) each adopt their usual interpretations. An atom, $p[t_1, \dots, t_n]$, evaluates to *true* (or alternatively, is said to be *satisfied*, written $I_I \models (p[t_1, \dots, t_n])[v]$) if and only if, when each term is evaluated to yield a constant (with anonymous variables adopting arbitrary constant values), there is a matching fact/tuple in the relation instance $I_I(p)$; otherwise, it evaluates to *false* (written $I_I \not\models (p[t_1, \dots, t_n])[v]$). The semantics of the universally quantified formula $\forall x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \varphi$ are roughly as follows: for all values of the variables x_1, \dots, x_i that satisfy the atom $p[t_1, \dots, t_n]$, the formula φ holds (i.e., evaluates to true). The semantics of the existentially quantified formula $\exists x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \varphi$ are similar, except that φ need only hold for one binding of the variables x_1, \dots, x_i that satisfies $p[t_1, \dots, t_n]$. The semantics of these expressions will be defined more precisely in later sections.

In the face of uncertainty, the evaluation becomes substantially more complicated; this case is addressed informally in Section 5.2.5, and more formally in the Appendix, in Section A.2.

5.2.3 Examples

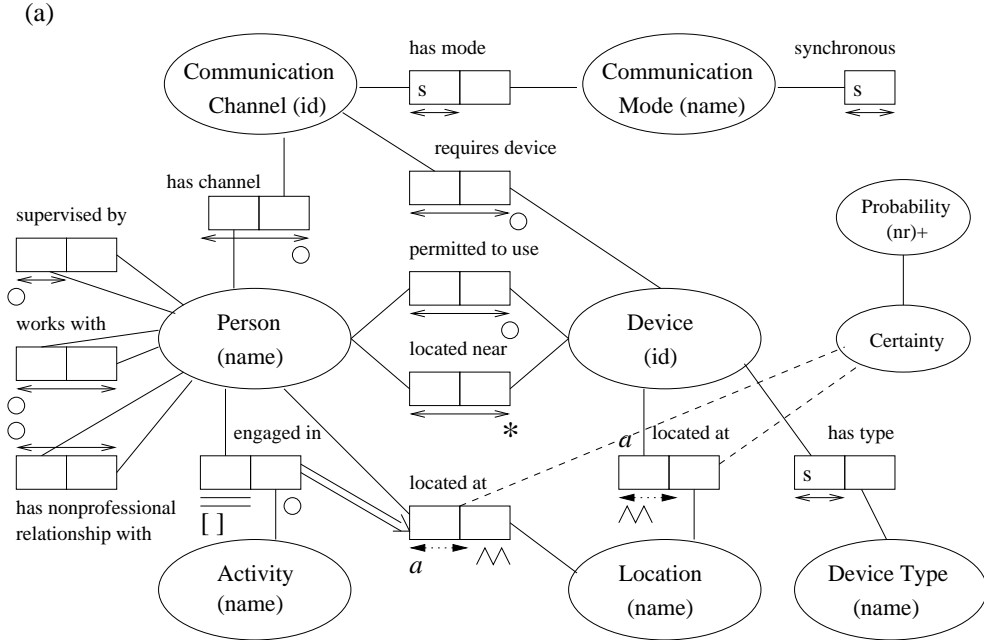
This section demonstrates the use of the situation abstraction to describe contexts, using some simple examples. These relate to the context-aware communication scenario outlined in Section 1.5, and assume the context model shown in Figure 5.1.

When selecting an appropriate communication channel for a given interaction between two or more users, parameters such as the activities of the users, the time of day and the available communication devices are relevant. Figure 5.2 lists a variety of example situations that can have a bearing on the choice of channel.

The first predicate in the figure performs a simple mapping from the *SupervisedBy* relation/fact type to a corresponding situation that describes the scenario involving a call from a supervisor to one of his/her subordinates. The next two predicates are similar, and represent the situations in which an interaction involves a pair of colleagues or a pair of non-professional acquaintances, according to the *WorksWith* and *NonProfessionalRelationship* relations, respectively.

The *WorkingHours* predicate demonstrates a simple use of built-in and user-defined functions. The *timenow* built-in function operates as described in the previous section, while the user-defined *workinghours* function determines whether the time and date returned by *timenow* fall within standard office hours (9am to 5pm, Monday to Friday, excluding public holidays).

The *LocatedAt* predicate illustrates the use of quality information to create a



* located near(p,d) iff located at(p, l1)
and located at (d, l2)
and l1 = l2

engaged in(p1,a) dependsOn located at(p2,l)
iff p1 = p2

(b)

<u>HasChannel(Person, CommunicationChannel)</u>	<u>DeviceLocatedAt(Device, Location, Probability)</u>
<u>PermittedToUse(Person, Device)</u>	<u>HasType(Device, DeviceType)</u>
<u>EngagedIn(Person, Activity, StartTime, EndTime)</u>	<u>HasMode(Channel, CommunicationMode)</u>
<u>SupervisedBy(Person, Supervisor)</u>	<u>RequiresDevice(Channel, Device)</u>
<u>WorksWith(Person1, Person2)</u>	<u>Synchronous(CommunicationMode)</u>
<u>NonProfessionalRelationship(Person1, Person2)</u>	<u>LocatedNear(Person, Device)</u>
<u>PersonLocatedAt(Person, Location, Probability)</u>	<u>LocatedAtDependents(FactID, DependentID)</u>

Figure 5.1: (a) A context model for the context-aware communication scenario. (b) The corresponding relations.

$SupervisorCall(caller, callee) :$
 $SupervisedBy[caller, callee]$

$ColleagueCall(caller, callee) :$
 $WorksWith[caller, callee] \vee WorksWith[callee, caller]$

$PersonalCall(caller, callee) :$
 $NonProfessionalRelationship[caller, callee] \vee$
 $NonProfessionalRelationship[callee, caller]$

$WorkingHours() :$
 $workinghours(timenow())$

$LocatedAt(person, place) :$
 $\exists probability \bullet PersonLocatedAt[person, place, probability] \bullet$
 $probability > 0.8$

$Occupied(person) :$
 $\exists t_1, t_2, activity \bullet EngagedIn[person, activity, t_1, t_2] \bullet$
 $(t_1 \leq timenow() \wedge (timenow() \leq t_2 \vee isnull(t_2))) \vee$
 $(t_1 \leq timenow() \vee isnull(t_1)) \wedge timenow() \leq t_2 \wedge$
 $(activity = \text{“in meeting”} \vee activity = \text{“taking call”})$

$CanUseChannel(person, channel) :$
 $\forall device \bullet RequiresDevice[channel, device] \bullet$
 $LocatedNear[person, device] \wedge PermittedToUse[person, device]$

$SynchronousMode(channel) :$
 $\forall mode \bullet HasMode[channel, mode] \bullet Synchronous[mode]$

$Urgent(priority) :$
 $priority = \text{“high”}$

Figure 5.2: Example situation predicates for the context-aware communication scenario.

precise definition of a situation in the face of uncertainty. This predicate deems a person to be located at a given place provided that a fact to this effect is present in the context instance, and the probability estimate associated with this fact exceeds 80%.

The *Occupied* predicate indicates whether a given person is currently engaged in an activity that generally should not be interrupted (“in meeting” or “taking call”), on the basis of the temporal *EngagedIn* relation. This predicate examines those activity facts for which the current time falls between the associated start and end times, or else does not precede the start time or succeed the end time in the case of activities that have no recorded end or start times, respectively.

CanUseChannel is satisfied for a given person, p , and communication channel, c , when all of the devices required in order to use c are located in close proximity to p , and p additionally has permission to use these devices.

The *SynchronousMode* predicate holds for a given communication channel provided that the mode of this channel (as recorded by the *HasMode* relation) is synchronous (indicated by its appearance in the *Synchronous* relation).

Finally, the simple *Urgent* predicate is satisfied whenever the *priority* variable has the value “high”.

Although a rich set of situations can be defined using predicates such as those described here, the true power of the situation abstraction comes from the ability to combine predicates arbitrarily to form increasingly complex situations, as described in the following section.

5.2.4 Combining situations

It is trivially observed from the definition of the valid situation formulas given in Appendix A that the formulas exhibit closure under logical negation, conjunction and disjunction. That is, provided that φ and ψ are both valid situation formulas, $\neg\varphi$, $\varphi \wedge \psi$ and $\varphi \vee \psi$ are likewise valid formulas, by definition. This property can be exploited to enable increasingly complex situation predicates to be defined recursively in terms of simpler predicates. Composite situations are defined using the following notation:

$$CS(v_1, \dots, v_n) : P$$

Here, CS is the name of the composite situation and P is an expression which combines one or more atoms over situation predicates using negation, conjunction and disjunction. Each atom is of the form $S(a_1, \dots, a_i)$, where $S(u_1, \dots, u_i) : Q$ is a previously defined predicate (ordinary or composite) and each a_j ($1 \leq j \leq i$) is either a constant in **dom** or one of the variables v_1, \dots, v_n .

The effect of combining predicates in this way is essentially identical to that of

combining the corresponding situation formulas to form an ordinary situation predicate. For example, the combined predicate $CS(a, b) : S_1(a) \wedge S_2(b)$, where S_1 and S_2 are non-composite predicates of the form $S_1(a) : \varphi$ and $S_2(b) : \psi$, evaluates identically to the non-composite predicate $S(a, b) : \varphi \wedge \psi$. A more complete discussion of the syntax and semantics of composite situation predicates appears in Section A.3.

The ability to combine situations is significant as it allows for reuse and the formation of complex predicates with relative ease. Returning to the context-aware communication scenario, the remainder of this section demonstrates the description of relevant aspects of the context using straightforward composite situation predicates formed from the simple predicates introduced in the previous section.

The scenario involves two distinct uses of context. The first is as follows (reproduced from Section 1.5):

Mark has finished reviewing a paper for Emma, and wishes to share his comments with her. He instructs his communication agent to initiate a discussion with Emma. Emma is in a meeting with a student, so her agent determines on her behalf that she should not be interrupted. The agent recommends that Mark contact Emma by email. Mark composes an email on the workstation he is currently using [channel = "mark@email.org"], and his agent dispatches according to the instructions of Emma's agent to her work email address.

The context involved in this scenario can be described in high-level terms as follows, using the predicates of Figure 5.2:

$$\begin{aligned}
 CS_1() : \\
 & \text{ColleagueCall}(\text{"Mark"}, \text{"Emma"}) \wedge \text{Occupied}(\text{"Emma"}) \\
 & \wedge \text{CanUseChannel}(\text{"Mark"}, \text{"mark@email.org"})
 \end{aligned}$$

The second usage case in the communication scenario is as follows:

A few minutes later, Emma's supervisor, Michelle, wants to know whether the report she has requested is ready. Emma's agent has been instructed that calls from Michelle have high priority, and decides that the query should be answered immediately. The agent suggests that Michelle telephone Emma on her office number [channel = "+61 7 3365 1111"]. Michelle's agent establishes the call using the mobile phone that Michelle is carrying with her [channel = "+61 4 1278 9999"].

Here, the relevant aspects of the context are captured by the composite predicate:

$$\begin{aligned}
 CS_2() : \\
 & SupervisorCall(\text{“Michelle”}, \text{“Emma”}) \wedge Occupied(\text{“Emma”}) \\
 & \wedge CanUseChannel(\text{“Emma”}, \text{“+61 7 3365 1111”}) \\
 & \wedge CanUseChannel(\text{“Michelle”}, \text{“+61 4 1278 9999”})
 \end{aligned}$$

It should be noted that any given real-world context corresponds to many alternative situational representations. The representation chosen by the user or the application developer for a given usage scenario will be determined by the task for which the context information is used and the subset of the context that requires emphasis for this particular task. The composition mechanism enables the selection and combination of the relevant aspects in a flexible fashion, such that situation definitions can be reused and redefined on-the-fly in response to changing resources and user requirements.

5.2.5 Handling uncertainty

The discussion of the situation abstraction has, to this point, implicitly assumed that the set of context information against which predicates are evaluated provides a sufficient basis for determining the absolute truth or falsity of a predicate, given an appropriate set of variable bindings. Clearly, this assumption is unrealistic in light of the imprecise and incomplete nature of context information. In this section, the assumption of certain context information is lifted, and the treatment of alternative facts and unknowns receives particular attention. The ability of the situation abstraction to support uncertainty, as outlined in the following paragraphs, is significant as the previously proposed predicate-based models of context have implicitly assumed the availability of perfect context information [10, 16].

The situation abstraction deals with uncertainty in a similar manner to a variety of relational query languages (such as SQL [186]), which employ three-valued logics in order to accommodate null values in facts. This approach is adopted as it is reasonably straightforward to implement and work with. When evaluating queries over relations, a third logical value, *possibly true* (sometimes also written in this thesis simply as *possibly*), is introduced when a query is satisfied only by creating arbitrary bindings for the null values that appear in facts. This can be interpreted as an unknown value, reflecting an inability to determine absolute truth or falsity from the available information.

A three-valued logic is applied to the situation abstraction as follows. An atom, $p[t_1, \dots, t_n]$ is viewed as *possibly true* for a context instance I_I and valuation of variables v when there is no fact in $I_I(p)$ that exactly matches the atom, but there

is a fact containing one or more null values that matches the atom once the nulls are replaced with arbitrary constant values in **dom**. In this case, I_I is said to partially satisfy $p[t_1, \dots, t_n]$ for v (written $I_I \vdash (p[t_1, \dots, t_n])[v]$ in the notation of Appendix A).

The three-valued logic also comes into play in the face of alternative facts, according to the interpretation set out in Section 4.6.9. Atoms over alternative relations are evaluated as follows. An atom $p[t_1, \dots, t_n]$ that fails to match any fact in the relation $I_I(p)$, even with nulls replaced with appropriate constants, evaluates to *false* as usual. Otherwise, if $p[t_1, \dots, t_n]$ matches a fact in $I_I(p)$, without reliance on replacement of nulls, *and* the matching fact has no alternatives, the atom evaluates to *true*. (Recall that a fact has alternatives if and only if there is at least one other fact that has identical values for the roles spanned by the alternative uniqueness constraint. This occurs whenever there is conflicting information present in the context instance.) Otherwise, if all matching facts have one or more alternatives or rely on appropriate bindings of nulls, $p[t_1, \dots, t_n]$ evaluates to *possibly true*.

The usual semantics of the logical connectives, \wedge , \vee and \neg , under a three-valued logic are adopted; these are summarised by the truth table in Table 5.1. The quantifiers are interpreted as follows. The universally quantified expression $\forall x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \varphi$ is satisfied, and yields the value *true*, provided that, for each instantiation of the variables x_1, \dots, x_i to constants in **dom** that satisfies the atom $p[t_1, \dots, t_n]$, the same instantiation satisfies φ . Otherwise, the expression is partially satisfied, and yields *possibly true*, whenever each instantiation of the variables to constants in **dom** that satisfies $p[t_1, \dots, t_n]$ also partially or fully satisfies φ . Otherwise, the expression yields *false*. The existentially quantified expression $\exists x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \varphi$ yields *true* when there is some assignment of the variables x_1, \dots, x_i to constants in **dom** for which the expression $p[t_1, \dots, t_n] \wedge \varphi$ yields *true*. Otherwise, the expression yields *possibly true* when there is an assignment of x_1, \dots, x_i to constants in **dom** for which $p[t_1, \dots, t_n]$ is *true* and φ is *possibly true*. Otherwise, the expression yields *false*. Again, the reader is referred to Section A.2 for a more formal treatment of these semantics.

It can be observed that, in the absence of uncertainty, the logical connectives and quantifiers preserve the usual semantics of standard boolean logic (with the exception that domain restriction is applied to the quantifiers). However, some intuitive results of standard logic no longer hold in the face of uncertainty. Two well-documented cases are the law of excluded middle ($x \vee \neg x$) and the law of non-contradiction ($\neg(x \wedge \neg x)$) [187]. Another type of anomalous behaviour, related to the alternative fact type, was described in Section 4.6.9 and is easily illustrated using an example. Suppose that the location of a person (Emma May) is known to be either room 98-122, 98-123 or 45-234, as in the relation instance shown earlier in

Table 5.1: Truth table for the three-valued logic.

p	q	$p \wedge q$	$p \vee q$	$\neg p$
<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>possibly</i>	<i>possibly</i>	<i>true</i>	
<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	
<i>possibly</i>	<i>true</i>	<i>possibly</i>	<i>true</i>	<i>possibly</i>
<i>possibly</i>	<i>possibly</i>	<i>possibly</i>	<i>possibly</i>	
<i>possibly</i>	<i>false</i>	<i>false</i>	<i>possibly</i>	
<i>false</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>possibly</i>	<i>false</i>	<i>possibly</i>	
<i>false</i>	<i>false</i>	<i>false</i>	<i>false</i>	

Table 4.7. The situation formulas

$$\begin{aligned}
& PersonLocatedAt[“Emma May”, “98-122”, _] \\
& \wedge PersonLocatedAt[“Emma May”, “98-123”, _] \\
& \wedge PersonLocatedAt[“Emma May”, “45-234”, _]
\end{aligned}$$

and

$$\begin{aligned}
& PersonLocatedAt[“Emma May”, “98-122”, _] \\
& \vee PersonLocatedAt[“Emma May”, “98-123”, _] \\
& \vee PersonLocatedAt[“Emma May”, “45-234”, _]
\end{aligned}$$

both evaluate to *possibly true*, rather than the more intuitive and informative values of *false* and *true*, respectively. The removal of these problems requires a substantially more complex interpretation of the relational and situational models of context, and falls outside the scope of this thesis. The investigation of multi-valued logics that preserve properties such as the laws of excluded middle and non-contradiction in the face of uncertainty has been ongoing for several decades without very satisfactory results.

In its favour, it can be shown that the three-valued logic that is adopted here never leads to contradictions; rather, the aforementioned problems can be characterised as losses of information (that is, results of *possibly true* in place of more definite *true* or *false* values). Provided that situation predicates are constructed with due care (that is, with a correct understanding of the semantics of third logical value), this need not be problematic.

5.2.6 Efficiency

The scale and resource limitations inherent in pervasive systems and the highly dynamic nature of context information imply that timely evaluation of situation pred-

icates is crucial. This section demonstrates the ability of the situation abstraction to meet this requirement while affording the expressive power to capture complex situations. Moreover, it argues that the expressiveness and efficiency of the situation abstraction compare favourably to those of the alternative predicate-based models of context mentioned in Section 5.2.1.

When evaluating the efficiency of the situation abstraction, the crux lies in the treatment of the quantified expressions; the remaining operators (logical, comparison and arithmetic) all have straightforward (constant time) evaluations. The evaluation of each atom, $p[t_1, \dots, t_n]$, is at worst linear in the number of facts present in the relation $I_I(p)$ ². The efficiency of the user-defined and built-in functions is a consideration that is left to the programmer to evaluate on a case-by-case basis.

The challenge associated with quantification rests in bounding the number of variable bindings that must be examined in the evaluation of the quantified expression. In ordinary predicate calculus, the number of bindings is potentially infinite (as, for example, in $(\forall x)(x > (x - 1))$, where x ranges over all of the integers). Such unbounded expressions are termed *unsafe* and their evaluation is either difficult or intractable. The usual way to eliminate this problem is to apply domain restriction, such that the values of the variables are assumed to fall within specified bounds. This is the approach employed by Ranganathan et al. [10] in their predicate-based model of context. While this approach resolves the problem of predicate safety, it remains unsatisfactory with regard to efficiency, as the domains involved will sometimes be large, and, where several variables are involved, the number of combinations that must be examined quickly becomes unmanageable.

The solution adopted here is to apply a variant of domain restriction which limits the values of the variables in a context-dependent fashion. When evaluating a quantified expression, the variables are immediately bound using an atom of the form $p[t_1, \dots, t_n]$. That is, the quantified variables, x_1, \dots, x_i assume exactly those values that satisfy $p[t_1, \dots, t_n]$ (and these can be computed trivially by a query on the relation instance $I_I(p)$).

This approach can be easily seen to avoid safety and efficiency problems, as the set of possible variable bindings is limited by the set of facts appearing in $I_I(p)$ (and this latter set is always finite, and often reasonably small). Further, the state explosion caused by the consideration of all possible combinations of variable bindings is prevented, as only those combinations appearing in $I_I(p)$ are significant.

The limitations (in terms of expressiveness) imposed on quantified expressions are offset against significant efficiency gains over the usual domain restriction approach, the infrequency with which broader forms of quantification are necessary, and the ability to embed these special cases within user-defined functions, where

²However, this can be easily improved upon using standard database techniques such as indexing.

issues related to safeness and efficiency can be considered on a case-by-case basis.

When compared to similar predicate-based models of context, the situation abstraction is easily seen to be both more flexible and more expressive despite its restricted forms of quantification. The model of Dey et al. [16, 111] offers no support at all for quantification, instead supporting only situations formed as simple combinations of conditions on atomic context attributes. As mentioned previously, the model of Ranganathan et al. [10] employs a form of domain restriction that is inefficient when domains are large or several variables are involved. Moreover, it adopts a flat and inflexible information model, which requires that all types of context be described by predicates of the form `Context(<ContextType>, <Subject>, <Relater>, <Object>)`. This rigid format is inappropriate for the representation of complex forms of information, such as historical and imperfect context information, as discussed in Section 3.3. Additionally, the model offers no support for user-defined or built-in functions.

5.3 Programming models

The situation abstraction developed in the previous section forms a natural basis for programming flexible context-aware applications. By programming an application's context-aware behaviour in terms of situations, rather than in terms of fine-grained facts, the application is effectively decoupled from the underlying fact-based context model. This allows the structure, representation and content of context information to be completely transparent to applications, and, by implication, to be changed with minimal impact. Moreover, the situation abstraction describes the classes of context that are relevant to applications in a high-level and application-neutral format, such that these classes can be modified, reused and recombined easily and on-the-fly without modification of the application source code, leading to highly dynamic and evolvable application behaviour.

The following sections describe two complementary programming models that follow naturally from the situation abstraction. The first (the triggering model) is event-based, invoking actions in response to significant context changes. In contrast, the second (the branching model) involves the embedding of context-dependent choices (branches) within the normal flow of application logic.

5.3.1 Triggering model

As mentioned previously, the triggering model has been widely explored in relation to adaptive and context-aware applications [23, 28–35]. The goal of this section is to present an overview of this well-known model and some of its realisations in context-aware systems, and then demonstrate that the situation abstraction forms

an excellent basis for a novel and powerful form of triggering (termed *situation-based triggering* or SBT), which offers advanced support for uncertainty. Significantly, the SBT model accommodates both incomplete and ambiguous context information, and, additionally, allows triggers to exploit relevant quality indicators.

Overview

The triggering model is event-based, and operates roughly as follows. Programmers (and potentially also users [28, 29]) create a set of rules, each consisting of an event and a corresponding action. These rules are placed in a triggering engine, which is responsible for detecting the occurrence of significant events and then invoking actions in accordance with the rules. Numerous variations on this basic model exist, one of the most common being the event-condition-action (ECA) model developed by early active database research [188, 189]. In this model, each rule is associated with a constraint that is evaluated following the event as an additional precondition on the execution of the action.

In context-aware systems, the event generally corresponds to a significant context change, while the action is an appropriate response to this event. For example, in the case of a context-aware museum guide, triggers can be associated with movement between exhibits, such that information presented to the user is always relevant to the closest exhibit. Similarly, reminder applications can use triggers to associate reminder notes (e.g., “pick up more milk”) with the situations in which they are applicable (“within 100m of a supermarket”).

Most realisations of context-aware triggering describe contexts in very simple terms (typically as simple boolean conditions on context variables, as in [28–30, 34]). However, the model employed by Yau et al. [31] in their context-sensitive middleware is somewhat more sophisticated, allowing temporal expressions such as $a \rightarrow b$ (denoting the succession of condition a by condition b). Schmidt et al. [23] propose an alternative model that is unique in that it removes the assumption of perfect context information by allowing certainty thresholds to be associated with triggers. Their triggers are created using the following primitives:

- if enter(v, p, n) then perform action(i)
- if leave(v, p, n) then perform action(i)
- if in(v, p, m) then perform action(i)

Here, v is a previously defined context, p is quality threshold expressed as a probability, i is the action to be executed and n and m are delays expressed in milliseconds. In the first two cases, i is performed n milliseconds after the entering/exiting of situation v is detected with a probability of at least p . In the third case, m is

the frequency with which i is executed while the situation v continues to hold with probability p .

The SBT triggering model can be viewed as a hybrid of the models of Yau et al. and Schmidt et al., incorporating the temporal features of the former model, and a variant of the event model (based on the entering and leaving of contexts) of the latter. The richness of the situation abstraction, and of the underlying fact-based model of context, affords significant expressive power to this triggering model. Imprecise context information is supported using the heterogeneous quality modelling approach described in Section 4.4.3, while alternatives and unknowns are accommodated using a three-valued logic as described in Sections 4.6.9 and 5.2.5. Accordingly, the SBT model provides an unprecedented degree of support for uncertain context information. Even the model of Schmidt et al., which accommodates context information of limited precision as described above, cannot handle unknowns and alternatives, while other models lack support for uncertainty in any form [28–34].

SBT adopts an ECA model, where:

- the event corresponds to a relevant situation change;
- the condition is a logical expression (formed from situation predicates) that must hold following the event and prior to the execution of the action; and
- the action consists of an arbitrary sequence of steps described in a programming language such as Java.

The formulation of events and conditions is described in the sections that follow.

Trigger events

In the SBT model, triggers are activated in response to relevant context changes. As contexts are described in terms of situations, SBT events represent transitions between situation states. The adoption of a three-valued logic in the evaluation of situation predicates trivially implies that three states are possible when evaluating a situation, $S(v_1, \dots, v_n) : \varphi$ against a valuation v for the variables v_1, \dots, v_n and a context instance, I_I . These states can be named as follows:

- In S (when S evaluates to *true* - or in the notation of Appendix A, $val(S, I_I, v) = true$);
- Possibly in S (when S evaluates to *possibly true* - i.e., $val(S, I_I, v) = possibly true$); and
- Not in S (when S evaluates to *false* - i.e., $val(S, I_I, v) = false$).

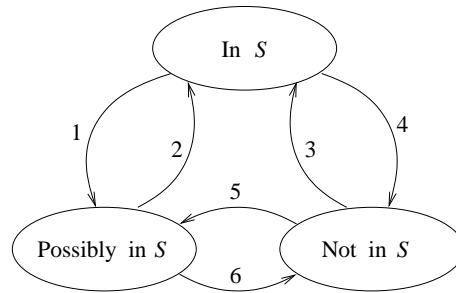


Figure 5.3: State transitions for a situation S .

There are six distinct state transitions, as shown in Figure 5.3. Triggers can be associated with any one or more of these transitions. Specifically, trigger events can be any of the following expressions:

- $EnterTrue(S)$ - transition 2 or 3
- $EnterFalse(S)$ - transition 4 or 6
- $EnterPossiblyTrue(S)$ - transition 1 or 5
- $ExitTrue(S)$ - transition 1 or 4
- $ExitFalse(S)$ - transition 3 or 5
- $ExitPossiblyTrue(S)$ - transition 2 or 6
- $TrueToFalse(S)$ - transition 4
- $TrueToPossiblyTrue(S)$ - transition 1
- $FalseToTrue(S)$ - transition 3
- $FalseToPossiblyTrue(S)$ - transition 5
- $PossiblyTrueToTrue(S)$ - transition 2
- $PossiblyTrueToFalse(S)$ - transition 6
- $Changed(S)$ - any transition

Trigger events can also take the form of:

- a sequence $e_1 \rightarrow \dots \rightarrow e_n$ of events, all of which must occur (in succession, in the order in which they appear) in order to activate the trigger; or
- a set $e_1 | \dots | e_n$ of alternative events, any one of which can activate the trigger independently of the others.

Here each e_i ($1 \leq i \leq n$) is an arbitrary event that may itself be a sequence or set of alternative events.

Some example events will be presented by way of illustration later in this section.

Trigger conditions

SBT conditions are logical expressions that must be fully satisfied (that is, *true* rather than merely *possibly true*) following the corresponding trigger event in order for the action to be executed. These are either the constant value *true* or an *extended predicate formula* containing arbitrary constants in **dom** but no variables. An extended predicate formula is a logical expression formed by combining one or more atoms, $S(c_1, \dots, c_n)$, using the logical operators (\wedge , \vee and \neg) and the two special functions, *possibly* and *possiblynot*. The special functions are used to remove uncertainty from formulas as follows. The formula *possibly*(x) is satisfied whenever the formula x is either fully or partially satisfied. Similarly, *possiblynot*(x) is satisfied whenever x is unsatisfied or only partially satisfied.³ A more formal treatment of these special functions and the extended predicate abstraction appears in Section A.4.

The following are all valid trigger conditions, expressed in terms of the situation predicates of Figure 5.2:

- *true*
- *LocatedAt*(“Emma May”, “45-234”)
- *possiblynot*(*LocatedAt*(“Emma May”, “45-234”))
- *LocatedAt*(“Emma May”, “45-234”) \wedge
 \neg (*possibly*(*Occupied*(“Emma May”)))

Some examples of complete triggers that can be formed using these conditions appear in the following section.

Examples

Figure 5.4 shows a variety of triggers that can be expressed using the situation predicates of Figure 5.2. The first three lines of each trigger (preceded by the keywords **upon**, **when** and **do**, respectively) specify the event, condition and action. The fourth line indicates the lifetime of the trigger; this is:

- **always** when the trigger applies indefinitely;
- **from** <start> **until** <end> when the trigger applies only between the <start> and <end> times;
- **until** <end> when the trigger expires after the specified the **end** time;
- **once** when the trigger expires upon the first execution of the action; or

³Note that *possiblynot*(x) is equivalent to *possibly*($\neg x$), but is included for convenience.

- `<n>` times when the trigger expires upon the n th execution of the action.

An optional fifth line, `after <delay>` interposes a delay of a specified number of seconds between the activation of the trigger and the execution of the corresponding action.

Actions are described in the examples of Figure 5.4 in natural language. However, in an implementation of the SBT model, these would be replaced by appropriate blocks of code.

The first trigger notifies the user (“Emma May”), 10 seconds after the conclusion of an important engagement (a meeting or call), of all recent missed calls.

The next trigger has the effect of diverting the user’s office telephone to voicemail whenever she leaves her office (“45-234”). This occurs whenever the state of the expression `LocatedAt(“Emma May”, “45-234”)` changes directly from *true* to *false*, or from *true* to *false* through the *possibly true* state.

The third trigger displays a traffic report to the user at the end of the working day, provided that the user is both located in her office and not occupied with a meeting or caller.

The remaining two triggers are closely related. Their combined effect is to display to the user a list of the day’s meetings at the beginning of the working day. The first handles the case in which the user is present in her office and not engaged in meetings or calls; in this case, the list is presented immediately. The second is concerned with the case in which the user is not known to be definitely in her office and unoccupied; when this situation arises, a new one-time trigger is created to display the information once the user is detected in her office and is known to be unengaged.

5.3.2 Branching model

The branching model is an alternative programming model in which context-dependent choices are inserted into the normal flow of application logic. Although this model has been widely used in the programming of context-aware applications, it has not received explicit recognition as an important programming model, and has generally been realised in a primitive form using complex if or case statements embedded within the application source code. These primitive branching mechanisms are unsatisfactory, as they tightly bind the context model to the application logic, and lead to applications that are inflexible and difficult to maintain. In many cases, details of the structure of the context model, the representation of context information or even the context gathering infrastructure are embedded in the code, rendering the modification of any of these extremely difficult. Moreover, the context-dependent choices supported by this approach are static, in that a given context always leads to the same action unless the source code is modified, recompiled and redeployed.

```

upon   EnterFalse(Occupied("Emma May"))
when   true
do     Notify of recent missed calls
always
after  10

upon   TrueToFalse(LocatedAt("Emma May", "45-234") |
      (TrueToPossibly(LocatedAt("Emma May", "45-234") →
      PossiblyToFalse(LocatedAt("Emma May", "45-234")))
when   true
do     Divert office phone to voicemail
always

upon   ExitTrue(WorkingHours())
when   LocatedAt("Emma May", "45-234")
       $\wedge \neg(\textit{Occupied}(\textit{"Emma May"}))$ 
do     Display traffic report
always

upon   EnterTrue(WorkingHours())
when   LocatedAt("Emma May", "45-234")
       $\wedge \neg(\textit{Occupied}(\textit{"Emma May"}))$ 
do     Display summary of day's meetings
always

upon   EnterTrue(WorkingHours())
when   possiblynot(LocatedAt("Emma May", "45-234"))  $\vee$ 
      possibly(Occupied("Emma May"))
do     Create one time trigger:
      upon   EnterTrue(LocatedAt("Emma May", "45-234")) |
            EnterFalse(Occupied("Emma May"))
      when   LocatedAt("Emma May", "45-234")
             $\wedge \neg(\textit{Occupied}(\textit{"Emma May"}))$ 
      do     Display summary of day's meetings
      once
always

```

Figure 5.4: Example triggers.

A flexible model of branching is proposed here as a remedy to these problems. This model, named *situation-based branching (SBB)*, removes the logic involved in context-dependent choices (and, by implication, the details of the particular context information upon which each choice depends) from the source code. Like triggers, this information is instead expressed in an application-neutral format, and is easily modified as needed to support the requirements of different users and the evolution of these requirements over time.

Motivation

The goal of SBB is to allow developers to incorporate context-dependent behaviour into applications in a straightforward fashion without the need for complex decision logic. SBB requires that developers identify the choices in application behaviour that should be influenced by the context, but does not require the link between choices and contexts to be specified. This link is made externally within a set of user preferences.

The branching model supports diverse classes of context-aware behaviour, including proximate selection and contextual information and commands, as described by Schilit et al. [33], as well as context-aware information retrieval, as described by Brown and Jones [48, 110]. Within the scope of the context-aware communication application that was described in Section 1.5, SBB can be used to support the choice of communication channels by agents; this usage is explored further in Chapter 7, in which an implementation of the communication application is described.

The use of preference information as the basis for context-dependent choices represents a novel and crucial feature of SBB. The importance of user modelling techniques to capture user desires and goals in context-aware systems is increasingly being recognised [118–122]. However, despite this growing recognition, very little research has pursued this approach. One exception, described in Section 3.5, is the use of learning algorithms by Byun and Cheverst [118, 119] to predict user behaviour in connection with (theoretical) context-aware applications such as intelligent personal assistants. The user models generated by the algorithms are exploited to support a variety of proactive behaviours, such as the generation of reminder notices when the due date for a library book draws near or when an office door is left open and the user is not likely to return within a short time. However, this approach does not allow users to explicitly specify or view their preferences, and may require lengthy periods of training before complex user requirements are adequately learned.

In some cases, user preference information has been regarded as a type of context information [15, 124, 125]. When this approach is followed, preference information is generally extremely limited (for example, describing the user's preferred language and interests).

The preference information used by SBB takes a somewhat different form. Preferences form a link between contexts (described in terms of situations) and context-dependent choices. That is, they allow users to express which choices are appropriate in which contexts in a high-level, application-neutral format. By separating this information from the application, the need to hardcode the basis for a choice into the application logic is removed. This leads to highly flexible behaviour that can be modified simply by editing user preference information, and places the ultimate control over choices in the hands of the user. The latter feature, in particular, is likely to be crucial to the user acceptance of applications that aim for a high degree of autonomy.

Overview

The problem addressed by SBB can be summarised roughly as follows:

Given a finite set of candidate choices, $A = \{a_1, \dots, a_n\}$, a context instance, I_I , and a set of user preferences, P , select a subset, $C \subseteq A$, of choices that are appropriate given both I_I and P .

SBB supports decisions over arbitrary sets of candidate choices, much like the triggering model supports arbitrary actions. In the case of the communication application presented in Chapter 7, the choices correspond to communication channels, whereas in the case of an information retrieval application they are likely to represent a set of alternative documents or document types.

The next part of this chapter develops a model of user preferences that supports this decision problem. Sections 5.4.1 and 5.4.2 present the requirements and basic design of the preference model, while Section 5.4.3 explains how preferences, like situations, can be combined to support increasingly complex behaviours. Finally Section 5.4.4 outlines a basic programming toolkit that supports the SBB programming model, and Section 5.4.5 demonstrates that the preference model is compatible with preference learning techniques.

5.4 The preference abstraction

5.4.1 Overview and requirements

The survey of preference and user modelling techniques that appeared in Section 3.5 demonstrated that a diverse array of preference models have been developed to address a variety of subtly differing decision problems. The SBB model presents yet another form of decision problem for which none of the existing preference models is completely appropriate. This section characterises the unique set of requirements

and challenges posed by SBB prior to the development, in the following section, of a novel preference model that is designed expressly to satisfy these.

The key requirements of the preference model are predominantly related to usability. The success of the model hinges, in particular, on the ease with which users can formulate their own preferences. The model must therefore be conceptually simple, and, as in the case of the situation abstraction, must allow users to employ composition to form increasingly complex expressions with ease. The latter requirement is particularly challenging in light of the possibility (or even likelihood) of conflicting preferences. The suitability of the preference model for use with automated preference elicitation and evolution techniques is likewise important, as these will be essential in complex systems to help lighten the burden placed on users to maintain accurate preference information.

In addition to meeting high standards of usability, the preference model must provide adequate expressive power. In particular, it must allow common policy concepts, including the concepts of prohibition and obligation as defined by deontic logic [190], to be easily captured. That is, users must be able to both forbid and require choices in certain contexts.

Section 3.5 surveyed several alternative preference modelling approaches, including vector-based, qualitative and quantitative approaches. Vector-based solutions, while appropriate for automated preference elicitation as shown in [132], require all of the dimensions over which preferences range to be expressed numerically. Thus, they are best suited to domains in which the parameters over which decisions are made are simple and homogeneous (such as document retrieval applications in which text documents are ranked solely on the basis of keywords), and are not considered further in relation to SBB.

The qualitative preference modelling approach offers greater expressive power than the quantitative approach, as it can express nontransitive preferences [127,130]. However, it does not offer a satisfactory way to combine preferences: the various composition approaches that have been explored in connection with this model are either overly simplistic, yielding unacceptably poor results [130], or are computationally expensive or even intractable [131]. As the combination of preferences and efficiency are both of crucial importance, SBB instead adopts the quantitative approach. Its preference model is most closely aligned with the quantitative model proposed by Agrawal and Wimmers [129], which was designed with explicit support for preference composition in mind. This model boasts the additional benefits of being both conceptually clean and providing a *veto* mechanism which can be used to express the concept of prohibition. As it stands, however, the model of Agrawal and Wimmers does not allow preferences to be dependent on context, nor does it support obligation. The preference model developed in the following section overcomes both of these shortcomings.

5.4.2 Preference model

The preference model proposed here, like that of Agrawal and Wimmers, is based on scoring. Each preference assigns to each candidate choice a score that is determined according to the context. Scores are either numerical values in the range $[0,1]$ or one of the special values \perp , $\bar{\perp}$ or $?$. As in the model of Agrawal and Wimmers, \perp represents a veto (that is, that the candidate to which the score is assigned should not be selected in the corresponding context), while $\bar{\perp}$ represents indifference or an absence of preference. The new score $\bar{\perp}$ is introduced to represent obligation (that is, that the candidate to which the score is assigned *must* be selected in the corresponding context), while $?$ represents an undefined score (signalling an error condition). The numerical scores capture relative desirability amongst candidates such that, if a candidate a_1 receives a lower score than a second candidate a_2 , then a_2 is regarded as preferable to a_1 . Similarly, equal scores imply comparable desirability.

Preferences have two components:

- A scope, c , which specifies the contexts within which the preference applies. Like trigger conditions, scopes are either the constant *true* or an extended predicate formula comprising one or more atoms of the form $S(t_1, \dots, t_n)$ (where each t_i is either a constant in **dom** or a variable in **var**), together with zero or more of the logical operators (\wedge , \vee and \neg) and special functions (*possibly* and *possiblynot*).⁴ The set of variables in c is denoted $var(c)$.⁵
- A score, s . This can be a simple value, or an arbitrarily complex arithmetic expression, possibly containing variables and built-in and user-defined functions, in addition to the standard operators ($+$, $-$, \times and \div) and the numerical and special constants ($\mathbb{R} \cup \{\perp, \bar{\perp}, ?\}$). As shown in Section 5.4.3, the names of preferences (or groups of preferences) can also appear as function parameters. The set of variables in s is denoted $var(s)$.

The value of s , when evaluated against a context instance I_I and a valuation v on the variables $var(s)$, is written $score(s, I_I, v)$ and must belong to the set $[0, 1] \cup \{\perp, \bar{\perp}, ?\}$.⁶

As a shorthand, preferences can be written as pairs $p = (c, s)$. $p.c$ denotes the scope of preference p , and $p.s$ the scoring expression.

The evaluation of a preference $p = (c, s)$ with respect to a context instance I_I and a valuation v for the variables $var(c) \cup var(s)$ is straightforward, and is defined

⁴Note that trigger conditions differ from scopes slightly in that variables *are* permitted in the latter.

⁵Using the definitions of *form* and *free* given in Appendix A, $var(c) = free(form(c))$

⁶When the scoring expression s evaluates to a value that falls outside this set, $score(s, I_I, v)$ defaults to $?$.

as follows:

$$rate(p, I_I, v) = \begin{cases} score(p.s, I_I, v) & \text{if } p.c \text{ holds for } I_I \text{ and } v, \text{ (i.e.,} \\ & I_I \models form(p.c)[v] \text{ in the notation of A.4)} \\ \perp & \text{otherwise} \end{cases}$$

That is, when the context satisfies the scope, $p.c$, the score is simply that defined by $p.s$. Otherwise, when the scope does not hold, the preference yields a score of indifference.

Some example preferences, which rate the suitability of communication channels, and are designed to be used by the communication application that has been used as a running example throughout this thesis, appear in Figure 5.5. The preference scope in each case begins on the first line, following the keyword **when**, while the line prefixed by **rate** yields the corresponding score. In all of these examples, the score takes the form of a simple value. A discussion of more complex scoring examples is deferred until the following section, in which the composition of preferences is addressed.

The first preference forbids the use of synchronous channels, such as telephone and video-conferencing channels, when the user does not have access to all of the requisite devices. Similarly, **p2** forbids the use of synchronous channels for interactions when the user is engaged in a meeting or call, excepting urgent calls and those initiated by a supervisor.

Preferences **p3** and **p4** together imply that synchronous channels are the preferred choice for urgent calls. **p3** assigns these the highest score of 1, while **p4** assigns all asynchronous channels (such as email and SMS) a score of 0.5.

Preferences **p5** and **p6** are both concerned with after-hours calls from colleagues. Provided that these are not urgent, asynchronous channels are greatly preferred (1) over synchronous channels (0.2).

Finally, preference **p7** prohibits the use of synchronous communication channels for non-urgent personal calls.

It is important to note at this point that the preference format shown in Figure 5.5 need never be exposed directly to users. Instead, users could select from standard preference profiles prepackaged with their applications, or manipulate a set of application-specific options (which are mapped automatically to appropriate preferences) through through customisation interfaces. In some applications, users may never specify preferences at all, as these are instead learned based on the actions or explicit feedback of the user.

As seen from these examples, individual preferences generally capture very narrow portions of a user's overall requirements. The true power of the preference model comes not from the use of individual preferences, but from the aggregation of

```

p1 =  when SynchronousMode(channel) ∧
        ¬CanUseChannel(callee, channel)
        rate ½

p2 =  when SynchronousMode(channel) ∧ Occupied(callee) ∧
        ¬Urgent(priority) ∧ ¬SupervisorCall(caller, callee)
        rate ½

p3 =  when Urgent(priority) ∧ SynchronousMode(channel)
        rate 1

p4 =  when Urgent(priority) ∧ ¬SynchronousMode(channel)
        rate 0.5

p5 =  when ColleagueCall(caller, callee) ∧ ¬WorkingHours() ∧
        ¬Urgent(priority) ∧ ¬SynchronousMode(channel)
        rate 1

p6 =  when ColleagueCall(caller, callee) ∧ ¬WorkingHours() ∧
        ¬Urgent(priority) ∧ SynchronousMode(channel)
        rate 0.2

p7 =  when PersonalCall(caller, callee) ∧ ¬Urgent(priority) ∧
        SynchronousMode(channel)
        rate ½

```

Figure 5.5: Example preferences for the context-aware communication application. These assign ratings to communication channels that are determined by the context, the participants in the interaction (*caller* and *callee*) and the priority.

preferences to form complex descriptions of user requirements, as well as the integration of preferences from a range of sources, including application default preferences, user-specific preferences, and organisational preferences and policies (such as those of the user's employer). Two methods that can be used to combine preferences are described in the following section.

5.4.3 Combining preferences

In pervasive computing environments, users can have many applications and very large sets of preferences. Techniques are required to structure preferences according to purpose and owner, as well as to support dynamic composition of preferences, promoting reuse. This section introduces preference sets and composite preferences in response to these requirements.

Preference sets serve as simple structuring mechanisms, allowing preferences to be grouped according to purpose (for example, by type of choice, application and user). When a preference belongs to only one group, its membership within this group can be declared by defining the preference within the scope of this group, as shown in Figure 5.6 (a). Alternatively, previously defined preferences can be grouped into sets as shown in Figure 5.6 (b). Finally, sets can be combined using set union (\cup), as shown in Figure 5.6 (c), as well as with set intersection (\cap) and difference (\setminus).

Choices in the SBB model generally need to account for many preferences, making up one or more entire preference sets. As described in the previous section, each preference generates a score independently of other preferences. Therefore, when evaluated against multiple preferences, a choice may be assigned scores that are widely varying, and possibly even conflicting. In order to reach a decision, the scores produced by all of the relevant preferences must be combined to produce a single result. This task is performed by composite preferences.

Composite preferences take the same form as the simple preferences that were previously described, with the exception that their scoring expressions are generally more complex, being required to capture the policies employed to combine the scores of the component preferences. Arbitrary arithmetic expressions are permitted; however, for simplicity, a set of default policies (taking the form of built-in functions) are supplied. These include the following:

- *average(P)*. This function computes the score for a candidate as the average of the numerical scores assigned by the preferences belonging to the set $P = \{p_1, \dots, p_n\}$. That is, $score(average(P), I_I, v)$ (where v is a valuation covering

(a)

```

SystemChannelPrefs = {
  p1 =   when SynchronousMode(channel) ∧
          ¬CanUseChannel(callee, channel)
          rate ½
}

DefaultChannelPrefs = {
  p2 =   when SynchronousMode(channel) ∧ Occupied(callee) ∧
          ¬Urgent(priority) ∧ ¬SupervisorCall(caller, callee)
          rate ½

  p3 =   when Urgent(priority) ∧ SynchronousMode(channel)
          rate 1

  p4 =   when Urgent(priority) ∧ ¬SynchronousMode(channel)
          rate 0.5
}

UserChannelPrefs = {
  p5 =   when ColleagueCall(caller, callee) ∧ ¬WorkingHours() ∧
          ¬Urgent(priority) ∧ ¬SynchronousMode(channel)
          rate 1

  p6 =   when ColleagueCall(caller, callee) ∧ ¬WorkingHours() ∧
          ¬Urgent(priority) ∧ SynchronousMode(channel)
          rate 0.2
}

OrganisationalChannelPrefs = {
  p7 =   when PersonalCall(caller, callee) ∧ ¬Urgent(priority) ∧
          SynchronousMode(channel)
          rate ½
}

```

(b)

```

SystemChannelPrefs = {p1}
DefaultChannelPrefs = {p2, p3, p4}
UserChannelPrefs = {p5, p6}
OrganisationalChannelPrefs = {p7}

```

(c)

```

ChannelPrefs = SystemChannelPrefs ∪ DefaultChannelPrefs ∪
UserChannelPrefs ∪ OrganisationalChannelPrefs

```

Figure 5.6: Preferences can be assigned to sets either by initially declaring them within the scope of a set as in (a), or defining sets explicitly in terms of previously defined preferences as in (b). Sets can be combined as in (c).

all variables in each preference p_i in P) is generally equivalent to

$$\frac{\sum_{i=1}^n nScore(p_i, I_I, v)}{\sum_{i=1}^n numerical(p_i, I_I, v)}$$

where the functions *numerical* and *nScore* are defined as follows:

$$numerical(p_i, I_I, v) = \begin{cases} 1 & \text{if } rate(p_i, I_I, v) \in [0, 1] \\ 0 & \text{otherwise} \end{cases}$$

$$nScore(p_i, I_I, v) = \begin{cases} rate(p_i, I_I, v) & \text{if } rate(p_i, I_I, v) \in [0, 1] \\ 0 & \text{otherwise} \end{cases}$$

Indifferent scores are effectively ignored by the calculation.

However, the presence of *any* score of veto (\natural) forces the combined score to be veto likewise, and similarly for obligations ($\bar{\lambda}$). In the presence of both vetoes and obligations, or an undefined score, the end result is undefined (?). Thus, the semantics of the *average* function can be fully defined as follows:

$$score(average(\{p_1, \dots, p_n\}, I_I, v) =$$

$$\left\{ \begin{array}{ll} \perp & \text{if } rate(p_i, I_I, v) = \perp \text{ for all } i \\ \frac{\sum_{i=1}^n nScore(p_i, I_I, v)}{\sum_{i=1}^n numerical(p_i, I_I, v)} & \text{if } rate(p_i, I_I, v) \notin \{\natural, \bar{\lambda}, ?\} \text{ for all } i \\ ? & \text{if for some } i, rate(p_i, I_I, v) = ?, \text{ or for some } i, \\ & j, rate(p_i, I_I, v) = \natural \text{ and } rate(p_j, I_I, v) = \bar{\lambda} \\ \natural & \text{otherwise, and } rate(p_i, I_I, v) = \natural \text{ for some } i \\ \bar{\lambda} & \text{otherwise} \end{array} \right.$$

- *wgtaverage*($\langle p_1, \dots, p_n \rangle, \langle w_1, \dots, w_n \rangle$). This is identical to the function *average*, except that each preference, p_i , is associated with a weight, w_i , between zero and one (inclusive), representing its relative importance. Here, one

denotes the highest importance and zero the lowest⁷. The semantics of this function remain as before, except that, in the absence of the special scores \natural , $\bar{\lambda}$ and $?$, and the presence of at least one numerical score, the score is computed as follows:

$$\text{score}(\text{wgtaverage}(\langle p_1, \dots, p_n \rangle, \langle w_1, \dots, w_n \rangle), I_I, v) =$$

$$\frac{\sum_{i=1}^n \text{nScore}(p_i, I_I, v) \times w_i}{\sum_{i=1}^n \text{numerical}(p_i, I_I, v) \times w_i}$$

- *override*(p_1, p_2). This function, in general, produces scores aligned with the preference p_1 . However, special scores produced by p_2 are allowed to override the scores of p_1 as follows:

$$\text{score}(\text{override}(p_1, p_2), I_I, v) =$$

$$\begin{cases} \text{rate}(p_1, I_I, v) & \text{if } \text{rate}(p_2, I_I, v) \notin \{\natural, \bar{\lambda}, ?\} \\ \text{rate}(p_2, I_I, v) & \text{otherwise} \end{cases}$$

- *as*(p). This simple function indicates that the score should be produced as defined by the preference p . That is:

$$\text{score}(\text{as}(p), I_I, v) = \text{rate}(p, I_I, v)$$

Further policies can be implemented as user-defined functions, as required.

It should be noted at this point that this preference model differs markedly from that of Agrawal and Wimmers in its approach to combining preferences. Rather than combining the scoring expressions themselves, it directly combines the *scores* that are generated by the individual preferences. This approach is much better suited to the SBB problem, given that preference sets can potentially be extremely large (meaning that combined preferences would be often be extremely complex and unwieldy), and, additionally, that the scoping of preferences greatly complicates the task of combining scoring expressions. The direct combination of scores, in contrast, usually represents a straightforward arithmetic computation.

The use of the four built-in functions to define composite preferences in terms of the preference sets previously defined in Figure 5.6 is illustrated in Figure 5.7. In (a), the individual preferences of each of the four sets, `SystemChannelPrefs`, `DefaultChannelPrefs`, `UserChannelPrefs` and `OrganisationalChannelPrefs`, are combined using the *average* function. This is not strictly necessary for the first and

⁷However, assigning a preference a weight of zero is semantically equivalent to the omission of this preference altogether.

last of these sets, as they each contain only one preference, but is desirable because it allows for the addition of further preferences to the sets in the future. Next, in (b), the results of combining the system, default and user preferences are themselves aggregated, such that the combined default preference receives a slightly lower priority (0.8) than the combined system and user preferences (both 1).

The combination of the result (p12) with the combined organisational preference (p11) occurs in a context-dependent fashion, as shown in (c). During working hours, the combined system, default and user preferences override the composite organisational preference, except when the latter produces a veto, obligation or undefined score. Outside of working hours, the organisational preferences are entirely disregarded.

Finally, (d) combines the two context-dependent functions defined in (c) to form the single preference, p15. The scopes of the two component preferences, p13 and p14, are such that, for any given context and set of variable bindings, exactly one of the corresponding scopes will be satisfied. By implication, at most one of the scores produced by these preferences will be non-indifferent. This is the score adopted by p15⁸.

The relationship between the two approaches to combining preferences is illustrated in Figure 5.8. The preferences defined in Figures 5.5-5.7 can be seen to form a tree structure, in which the leaf nodes correspond to the simple preferences, and the composite preferences form the internal and root nodes. Thus, the use of composite preferences supports the construction of arbitrarily deep preference hierarchies, in which the nodes towards the top capture increasingly complex requirements. Note that, although the hierarchy in this case takes the form of a tree, this need not be the case. Similarly, a preference repository need not contain only one preference hierarchy - many hierarchies may be present, supporting a variety of choice types, users, applications, and so on.

Preference sets, shown in the figure as dashed rectangles, aim to make preference repositories more manageable by allowing preferences to be logically grouped. This grouping is generally based on factors such as choice type, user and application. In the figure, the preference sets involve only the simple (leaf) preferences, but this is not mandatory; sets consisting entirely of composite preferences are also permitted, as are mixed sets. Likewise, preferences can belong to zero or more sets, and preference sets can overlap arbitrarily.

5.4.4 Programming with preferences

Having characterised the preference model in the previous sections in some detail, it now remains to clarify the role of preferences in relation to the SBB model. In

⁸Clearly, if both p13 and p14 yield indifferent scores, the result of p15 will likewise be indifferent.

(a)

```

p8 =   when true
       rate average(SystemChannelPrefs)

p9 =   when true
       rate average(DefaultChannelPrefs)

p10 =  when true
       rate average(UserChannelPrefs)

p11 =  when true
       rate average(OrganisationalChannelPrefs)

```

(b)

```

p12 =  when true
       rate wgtaverage(< p8, p9, p10 >, < 1, 0.8, 1 >)

```

(c)

```

p13 =  when WorkingHours()
       rate override(p12, p11)

p14 =  when ¬WorkingHours()
       rate as(p12)

```

(d)

```

p15 =  when true
       rate average({p13, p14})

```

Figure 5.7: Combining the preference sets of Figure 5.6. In the composite preferences shown in (a), the scores produced by the system, default, user-defined and organisational preference sets are combined using *average*. Next, in (b), the scores of the combined system, default and user-defined preferences are aggregated using *wgtaverage*, such that the default preferences are given 80% of the weight of both the system and user-defined preferences. In (c), the result is combined in a context-dependent fashion with the combined organisational preferences. Finally, (d) combines the two preferences defined in (c) to yield a single aggregated score for any given context.

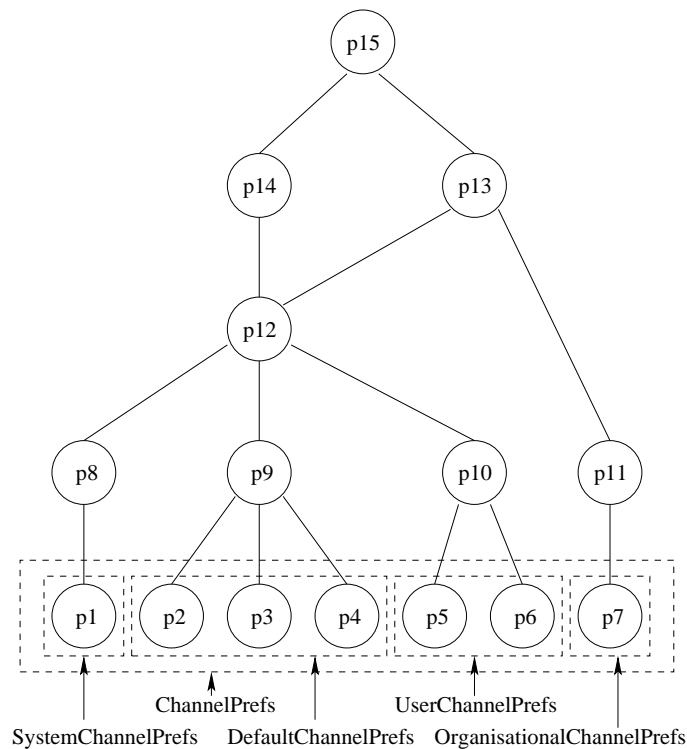


Figure 5.8: The hierarchy of preferences defined by Figures 5.5-5.7.

this section, the development of a set of programming tools that can be used to realise the branching model is briefly discussed. A more detailed discussion of the implementation of the SBB and preference models is, however, deferred until the following chapter.

The task involved in SBB was characterised in Section 5.3.2 as a decision problem in which a set of available choices is narrowed according to the context and the preferences of the user, so that only the most appropriate choices are selected. In some circumstances, the goal is to select exactly one choice (as, for example, in the case of a communication agent that has the task of selecting the most appropriate communication channel for an interaction on behalf of the user). In other situations, the goal may be to select all highly rated choices, or a fixed number of the best-rated choices (as in the case of a document retrieval application that returns the best matches according to the context and requirements of the user).

The main methods of a Java programming toolkit that supports these and other variants of the choice problem are illustrated in Figure 5.9. These are briefly characterised in the remainder of this section.

The Java methods support many different variants of the branching problem. In each case, the programmer specifies a set of candidate choices, the preference that is used to generate scores for the candidates, and a valuation that specifies appropriate bindings for all of the variables that appear in the preference (including all

```

Scores rate(Choice[] c, Preference p, Valuation v);
Scores rate(Choice[] c, Preference p, Valuation v, Context cx);

Choice selectBest(Choice[] c, Preference p, Valuation v);
Choice selectBest(Choice[] c, Preference p, Valuation v,
                  Context cx);
Choice[] selectBestN(int n, Choice[] c, Preference p,
                    Valuation v);
Choice[] selectBestN(int n, Choice[] c, Preference p,
                    Valuation v, Context cx);
Choice[] selectAbove(Score threshold, Choice[] c, Preference p,
                    Valuation v);
Choice[] selectAbove(Score threshold, Choice[] c, Preference p,
                    Valuation v, Context cx);
Choice[] selectMandatory(Choice[] c, Preference p, Valuation v);
Choice[] selectMandatory(Choice[] c, Preference p, Valuation v,
                        Context cx);

void invokeBest(Choice[] c, Preference p, Valuation v,
               Action default);
void invokeBest(Choice[] c, Preference p, Valuation v,
               Action default, Context cx);
void invokeBestN(int n, Choice[] c, Preference p, Valuation v,
                 Action default);
void invokeBestN(int n, Choice[] c, Preference p, Valuation v,
                 Action default, Context cx);
void invokeAbove(Score threshold, Choice[] c, Preference p,
                 Valuation v, Action default);
void invokeAbove(Score threshold, Choice[] c, Preference p,
                 Valuation v, Action default, Context cx);
void invokeMandatory(Choice[] c, Preference p, Valuation v,
                    Action default);
void invokeMandatory(Choice[] c, Preference p, Valuation v,
                    Action default, Context cx);

```

Figure 5.9: Selected methods of a programming toolkit that implements the SBB model.

component preferences). The preference is almost always a complex composite preference which combines a large, dynamically defined set of preferences from multiple sources (such as the composite preference `p15` of Figure 5.7). When no context is explicitly specified, the preference is evaluated against the current context stored in a context repository. The specification of a different context allows hypothetical contexts to be considered, including past and predicted contexts. The utility of such “pretended contexts” is demonstrated by Brown [29]. He cites an example in which a tourist wishing to retrieve information on an attraction not yet visited using a context-aware tour guide is able to do so by pretending to be at the location of interest by overriding the relevant context fields.

The onus lies on the programmer and the producer(s) of preference information to ensure that the supplied valuation covers all of the variables that appear in the preference and, similarly, to ensure that the preference itself is well-formed with respect to the context instance (specifically, that the scopes of the top-level preference and all of its component preferences conform properly to the extended situation abstraction, and that scoring expressions are similarly well-formed). Runtime exceptions result when these conditions are not met.

The toolkit offers three broad usage styles. In the first, scores are generated for a set of choices, and it remains the task of the programmer to determine the actions to be carried out in light of these. This style is supported by the two variants of the `rate` method. The return value in each case is a `Scores` object which maps each candidate choice to a score.

The second usage style is supported by the `select*` methods. These return one or more choices that match specified requirements. `selectBest` returns the single choice which is assigned the highest numerical score. If there are several such choices, one of these is selected arbitrarily. Alternatively, if no choice is assigned a numerical score, the return value is null. `selectBestN` works identically, except that it returns the n best-ranked choices, where n is determined by the first parameter. When no choices are assigned numerical scores, the result is null, as before; otherwise, the n highest-rated choices are returned, provided that at least n choices receive numerical scores. When fewer than n (and greater than zero) choices receive numerical scores, the return value is made up of *all* of these choices. `selectAbove` returns all of the choices that receive numerical scores at or above a specified threshold. Again, if there are no such choices, a null value is returned. Finally, `selectMandatory` returns all choices that receive obligation scores ($\bar{\wedge}$), or the value null if there are no such choices.

The remaining `invoke*` methods are almost identical to the `select*` methods, except that, instead of returning the selected choices, they automatically invoke the actions associated with these. The action for each `Choice` object is embedded within its `invoke` method. A default action is supplied to each of the toolkit’s `invoke*`

methods, and this is executed in the case that there are no suitable choices for invocation.

In Section 6.3.5, the implementation of the above methods as a complete programmer's toolkit is briefly outlined, while Section 7.11 discusses the implementation of a prototypical context-aware application using the toolkit.

5.4.5 Learning preferences

Owing to the potential scale and complexity of pervasive computing environments, minimisation of the effort demanded of users to create and maintain preference information is an important consideration. It is imperative that application developers supply default preference information that users can adopt without modification if required; additionally, learning algorithms that derive preference information from the behaviour and feedback of the user have been proposed by some researchers as an important feature of context-aware software [118, 119]. While a detailed investigation of appropriate learning techniques is clearly outside the scope of this thesis, this section demonstrates that such techniques are compatible with the preference model that was outlined in Sections 5.4.1 - 5.4.3.

There are at least two different approaches that can be used to evolve preferences in response to user feedback. The first involves the use of weighting to promote or reduce the importance of individual preferences in a composite preference according to user feedback. In its simplest form, this approach involves using negative user feedback (provided after an inappropriate choice is made) to decrease the weighting of preferences that assigned a high score to the choice, or to increase the weighting of preferences that assigned a low score. This type of weighting scheme is trivially supported by using the *wgtaverage* function introduced in Section 5.4.3 to construct an appropriate composite preference.

In the second approach, entirely new preferences are generated in response to feedback. For example, if the user rejects a choice, a new preference, having a scope that describes the context of the choice and a low score (or a veto) is incorporated into a relevant preference set. This approach is more flexible than the approach based on weighting, as the latter is required to work entirely with the pre-specified preferences, which may not cover all user requirements. However, this benefit is offset by the fact that learning can be very slow. The preference change that results from each instance of user feedback is generally only observable in the narrow context for which the feedback is given, meaning that rules that are applicable in broad contexts may require many rounds of feedback before they are derived in reasonably general forms.

These learning approaches are not mutually exclusive, and can be employed together in a scheme that combines the benefits of the two individual solutions.

5.5 Summary of contributions

The literature survey of Chapter 3 demonstrated that the bulk of recent context-awareness research has focused on the development of infrastructural support for context gathering and, to a lesser degree, on the modelling and management of context information. In contrast, the programming issues associated with context-awareness have received little attention. The few exceptions have mainly been limited in scope to narrow application domains. The goal of this chapter has been to begin to redress this problem. Progress towards this goal was made in three areas.

First, the situation abstraction was developed as a means to describe contexts in general and high-level terms. As discussed in Section 5.2, this abstraction masks the details of the context representation, allows contexts to be redefined as the underlying context model and user requirements change, and supports composition, such that situation descriptions can be reused and recombined to describe increasingly complex contexts in a straightforward fashion. Moreover, it was shown in Sections 5.2.5 and 5.2.6 to accommodate uncertain context information and to support efficient evaluation.

Second, complementary programming models, founded on the situation abstraction, were developed. The first model (SBT) was demonstrated to support powerful triggering expressions and to offer advanced support for uncertainty in comparison to similar models employed in adaptive and context-aware systems. The second model (SBB) was proposed as a novel replacement for the traditional if and case statements used to embed context-dependent choices into application logic. SBB relies on the use of preference information to reach a decision on the best choice(s) for a given context. The advantages of this model were shown in Section 5.3.2 to be threefold. First, SBB is extremely flexible, as the preference information used to support choices can be modified on-the-fly and personalised to meet the requirements of individual users. In contrast, choice logic that is embedded in application code is largely static. Second, SBB enables complete decoupling of context-aware applications from their context models. Thus, it allows context models and context gathering/management infrastructure to be modified independently of any applications that use them. This is important as the physical and computing environments through which users and applications migrate may vary greatly in the level of instrumentation they provide for context sensing, and, by implication, in the set of context information that is available. Third, the SBB model addresses the usability challenges associated with the programming of autonomous, context-aware software, as outlined in Section 5.1. Importantly, it affords users with a considerable degree of control over the actions of their applications.

Finally, a preference model appropriate for use with SBB was developed in Section 5.4. This model offers unique support for context-dependence (that is, variation

of scores assigned by preferences according to the context) and uncertainty. Moreover, it addresses key usability challenges by (i) allowing preferences to be specified in terms of simple, fine-grained expressions, which can be easily combined incrementally to support increasingly complex user requirements, and (ii) allowing for the use of automated preference elicitation techniques to reduce the burden on users to maintain accurate preference information. Importantly, the model also supports the concepts of obligation and prohibition, which allow users to formulate policies that expressly force or forbid choices in certain contexts.

The application-neutrality of all of the abstractions presented in this chapter, and consequent potential for sharing of situation predicates, triggers and preferences amongst applications, can also support a very desirable degree of consistency and unison between the actions of diverse applications belonging to a single user. This feature can be used to achieve the type of coordinated behaviour (preventing unstable or cascading adaptations) that Efstratiou et al. [75] argue is crucial in context-aware systems.

The practical utility of the situation abstraction and the branching and preference models is demonstrated in the following two chapters⁹. Chapter 6 develops the programming and modelling techniques into a software architecture and developer's toolkit for context-aware systems. Chapter 7 illustrates the utility of the theoretical models developed in this and the previous chapter, together with the software infrastructure presented in Chapter 6, in the development of a prototypical context-aware communication application.

5.6 Discussion and future work

Although the programming abstractions and models presented in this chapter were developed with the challenges of Section 2.2 in mind (heterogeneity, uncertainty, temporal issues, and dependencies), there is scope for improved support for the temporal and quality aspects of context information. In its current form, the situation abstraction offers a limited degree of support for temporal constraints. Simple constraints can be captured using basic techniques such as comparison of timestamps (see, for example, the *Occupied* predicate defined in Figure 5.2). Advanced constraints can also be represented using appropriate user-defined functions. However, an interesting avenue for future research would be the extension of the situation abstraction to include built-in support for temporal logic expressions such as $\diamond x$ (eventually x is true), $\Box x$ (always x is true) and $x \rightsquigarrow y$ (whenever x is true, y will eventually become true) [191], where x and y are atoms over temporal fact types.

⁹Note that the triggering model is considered only minimally in the remainder of this thesis as its utility to context-aware and adaptive applications has been widely demonstrated in the past (see, for example, [23, 28–35]).

Additionally, the current semantics of the situation abstraction (as set out in Appendix A) mirror the interpretation of the fact-based model outlined in Section 4.6.9. As discussed in Sections 4.6.9 and 5.2.5, this interpretation leads to sometimes unexpected and imprecise results (*possibly true* values when more definite *true* or *false* results are possible) in some cases involving alternative fact types or unknowns. The investigation of alternative interpretations remains a challenging topic for future work. It is likely that the solutions to the aforementioned problems will incur significant penalties in terms of complexity.

Finally, the preference model set out in Section 5.4 allows arbitrary user preferences to be captured, but further research is required to determine how user goals and desires can be effectively mapped to this model. The role of HCI methodologies, such as cooperative and consultative design processes and observational techniques [192] in identifying, evaluating and fine-tuning preferences remains an important topic for future investigation, as accurate and complete preference information is imperative to the utility of context-aware applications. Similarly, the use of learning techniques for automated preference elicitation and evolution (e.g., [132,193,194]) in connection with the preference model falls outside the scope of this thesis, but represents an interesting research topic.

Chapter 6

An architecture for context-aware pervasive systems

Chapters 4 and 5 presented theoretical foundations for the construction of context-aware systems, including a fact-based context modelling approach, and situation and preference abstractions that support programming models based on triggering and branching. In this chapter, the development of a software architecture built upon these foundations is considered. The design of this architecture is inspired by the need to overcome inherent challenges in pervasive computing systems related to autonomy, resource limitations, scalability and the dynamic nature of computing environments and user requirements, as outlined in Section 1.1.

The structure of this chapter is as follows. Section 6.1 introduces the design of a layered architecture for context-aware pervasive systems. Section 6.2 demonstrates the design features of this architecture that enable it to address the challenges introduced in Chapter 1. Section 6.3 briefly presents the design of a lightweight software infrastructure that implements this architecture. As the implementation focuses on the higher layers of the architecture (and most particularly, on the realisation of the programming models in the form of a programming toolkit), and has not been optimised for scalability, Section 6.4 outlines a variety of avenues for future work to address the current simplifications.

6.1 An architecture for context-aware systems

The architecture, illustrated in Figure 6.1, follows a layered approach to achieve a loose coupling of components and clean separation of concerns. The lowest layers assume responsibility for context gathering, reception and management. Next, the query layer offers an interface that allows the context to be queried in terms of the fact and situation abstractions characterised in the previous two chapters. The adaptation layer is responsible for managing a common pool of situation, preference and

trigger definitions, and evaluating these dynamically to support application adaptation. Finally, the application layer encompasses a set of programming tools that developers of context-aware applications can use to program software that exploits the services of the lower layers.

The following sections describe the functions and interactions of the layers in more detail, together with a variety of open research challenges that are important in a full implementation of the architecture, but beyond the scope of this thesis. A discussion of the extent to which the layers have been implemented is deferred until Section 6.3.

6.1.1 Context gathering layer

The context gathering layer is responsible for the acquisition of sensed context information from physical sensors (such as positioning devices) and logical sensors (such as software on a user's workstation that monitors activity taking place on the keyboard). This layer takes the form of a distributed network of sensors and processing components, structured in a hierarchical organisation resembling that of the Context Toolkit [95] or Solar context gathering infrastructure [22,100]. As the data derived from the sensors is generally far removed from the level of abstraction employed by the context management layer (atomic facts about entities), processing occurs within a variety of distributed software components to raise the level of abstraction and perform fusion of data from multiple sensors. In the terminology of the Context Toolkit, these components are *interpreters* and *aggregators*. Sensor data may pass through a chain of processing components before arriving at the context reception layer, where it is integrated into the context repository.

The interaction between the context gathering and reception layers is bidirectional. In the most common case, updates are delivered asynchronously to the reception layer by the context gathering layer. However, the reception layer will at times also actively interrogate the context gathering infrastructure in order to satisfy queries from the higher layers. Whether the active or passive approach should be favoured for a given type of context information is dependent on its characteristics (for example, how frequently the information changes), the expense associated with invoking the corresponding sensors and the ratio of updates to queries.

Two key concerns of the context gathering layer are scalability and routing of context information, due to the potentially large numbers of sensors involved in complex pervasive systems, and the often high frequency of sensor output. The placement of appropriate processing components is crucial. These should be employed to perform data reduction techniques, in order to distill streams of sensor output into discrete and relevant events. Likewise, the transport mechanism used for communication between sensing and processing components, and between these

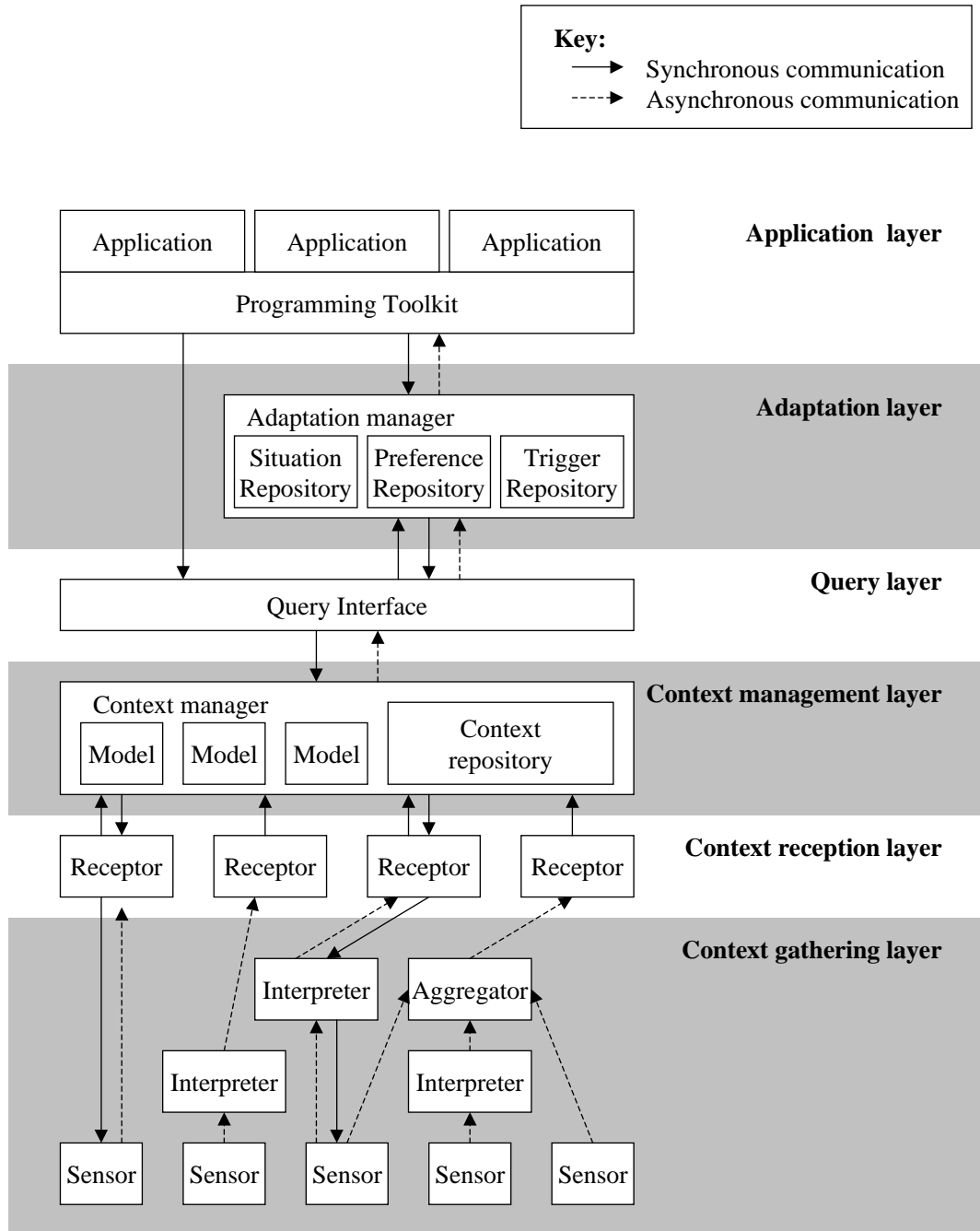


Figure 6.1: Architecture.

and the context reception layer, must be scalable. In order to address this concern, use of the Elvin event notification service [195] is proposed. Elvin implements a content-based routing scheme whereby notifications are routed to consumers on the basis of the content of the notification message, rather than any form of explicit addressing. The benefits of this approach are twofold. First, a very loose coupling between components is possible, as it is not necessary for the context management system (or interpreters and aggregators) to explicitly discover the appropriate sensors for gathering the required context information (which would inevitably change over time as a result of mobility and evolution of the sensor network). This loose coupling leads to greater tolerance of failures, including disconnections and component failures. Second, content-based routing is a highly scalable solution for distributed sensor networks, as events are forwarded only to relevant processing and context management components (and, indeed, need not be forwarded at all when there are no listeners).

6.1.2 Context reception layer

The context reception layer forms the interface between the context gathering and management layers. Its main role is to perform a translation between the heterogeneous information produced by the sensing and processing components of the lower layer, and the fact abstraction of the management layer. Each fact type is represented in the reception layer by a component that creates the appropriate event subscriptions, and, optionally, formulates queries when the active update model is used¹.

As reception components will often need to integrate context information from a variety of sources, conflict resolution may be required. This is illustrated using the example of location information. User locations can be tracked using special positioning devices, wireless networking information and swipe card logs. However, as individual readings are often imperfect or incorrect (for instance, when the user is not carrying his/her positioning device), a history of readings must be examined and reconciled in order to obtain an accurate location estimate.

As described in Chapter 4, there are two distinct approaches to dealing with conflicts. When inserting data into ordinary fact types, conflicts must be completely resolved at the reception layer. In contrast, the alternative fact types introduced in Section 4.4.4 permit conflicts to remain in the context repository. However, some conflict resolution at the reception layer may be desirable even in the latter case to remove improbable or impossible readings.

In addition, some of the processing needed to raise the level of abstraction of

¹Like the asynchronous notifications used in the passive model, these are routed by the Elvin event notification service

context information to match that of the fact-based context model may be deferred to the reception layer; however, as the reception layer will generally be located with the context management infrastructure, rather than with the sensors, data reduction should always be performed earlier in order to minimise bandwidth requirements.

6.1.3 Context management layer

The context management layer acts as a repository of context information that serves many context-aware applications. Its representation of context is based on the fact-based modelling approach introduced in Chapter 4 (or, more precisely, the relational mapping described in Section 4.6).

As each application may have its own model of context (that is, its own set of fact types and corresponding constraints), the management system is required to maintain many different models and potentially large quantities of information. The management system stores a common pool of context information, as well as a set of metadata corresponding to each distinct context model. The metadata characterises the set of fact types (relations) and constraints for each model, as outlined in Section 4.6.8, and additionally provides a mapping of context information from the common pool to a view that conforms to the model.

The scalability challenges faced by the context management layer are significant, owing not only to the volume of information to be managed, but also to the highly dynamic nature of many types of context information and the need for rapid query responses in order to support timely adaptation to context changes by applications. The use of a common pool of context information, rather than a separate repository for each context model, is important as it removes the need to duplicate shared context information, thereby reducing storage requirements and the overhead associated with maintaining consistency among the separate repositories. The scalability challenges are further addressed by appropriately distributing the context management system to allow context information to be located near the application(s) that require it, employing active rather than passive update techniques for some types of rapidly changing data (particularly when updates are much more frequent than queries), and adopting solutions of active and real-time database systems, such as scheduling strategies that ensure queries and updates are timely and temporally consistent [140]. Distribution introduces further challenges, as queries must be performed on distributed data, which may be incomplete in the face of disconnections and failures.

In addition to maintaining context information and responding to queries, the context management layer is responsible for enforcing the variety of constraints described in Chapter 4 and supporting the special fact types. For example, it is required to enforce the various types of uniqueness constraint, compute derived fact

types, maintain the histories of context information represented by temporal fact types (and enforce their corresponding temporal constraints) and manage dependencies². The context management layer must also allow users and applications to easily insert into, update and browse profiled fact types.

As discussed in Section 4.7, the gathering and storage of context information introduces a variety of privacy issues. The context management layer assumes the primary responsibility for managing and enforcing the privacy policies that are needed to govern the gathering, storage and querying of context information. However, in order to be effective, the policies must also be implemented within the context gathering, context reception and query layers.

6.1.4 Query layer

The query layer provides an interface that allows applications (as well as the adaptation layer) to easily formulate queries on a context model using the fact and situation abstractions. The query interface is a component that is generally local to the machine on which the client executes, and functions analogously to a stub in distributed architectures such as CORBA [196]. That is, it accepts queries from applications or the adaptation layer, translates these into the native query language of the context management system, and dispatches them (generally via a remote invocation) to the appropriate interface of the context management system. In the latter task, the query interface must handle issues of distribution, including disconnections and component failures, using strategies such as caching of query results and hoarding of frequently required context information, together with appropriate exception handling and reporting.

Three classes of query are supported by the query interface, as illustrated in Figure 6.2.

Fact queries are simple, atomic queries applied to a single fact type. These are either:

- *logical queries*, as in the first example in Figure 6.2 (a), which return one of the values *true*, *false* or *possibly true* according to the interpretation introduced in Section 4.6.9; or
- *binding queries*, as in the second example in Figure 6.2 (a), which contain free variables that become bound as a result of the query³.

²For example, by actively triggering updates of the dependent facts in response to changes in the facts they depend upon.

³This binding is somewhat complicated by the uncertainty associated with nulls and the alternative fact type. A binding of variables to values is made for each tuple in the relevant fact type for which none of the tuple's values for the variable attributes are null, and, for each of the remaining attributes, (i) the tuple's value is identical to specified value; or (ii) the tuple's value is null; or (iii) the anonymous variable is specified.

Both types of fact query can contain the anonymous (*don't care*) variable, which is represented by the underscore character as shown in the final example of Figure 6.2 (a).

Situation queries are expressed in terms of the extended predicate formulas introduced in the previous chapter. Some examples are illustrated in Figure 6.2 (b). These examples rely on situation predicates (*SynchronousMode*, *CanUseChannel*, *ColleagueCall*, *WorkingHours* and *Urgent*) defined in Figure 5.2. In order to evaluate the formulas, the query interface retrieves the required predicates from the situation repository maintained by the adaptation layer (and then caches these for use in subsequent queries). Situation queries may contain variables, as in the second example of Figure 6.2 (b); however, a binding of these variables must be supplied in order to perform the evaluation. Like logical fact queries, situation queries always yield one of the values *true*, *false* or *possibly true*.

Event queries are asynchronous queries, taking the form of a trigger event (as described in Section 5.3.1), which characterises a specific context change. Two examples are shown in Figure 6.2 (c). Instead of receiving an immediate response, as in the case of fact and situation queries, these result in the generation of asynchronous notifications when the corresponding events occur. When using this type of query, a mode and listener must be specified. The mode describes the persistence of the query (e.g., until the event has occurred once or n times, or until the query is explicitly revoked). The listener is the object to which notifications are delivered.

In general, direct *fact* queries by applications to the query layer should be minimised, with context-aware behaviour instead being incorporated using the triggering and branching models, or using situation and event queries sparingly when necessary. This approach minimises the level of coupling present between the application and its context model, and ensures that the latter can be modified without a corresponding modification of the application's source code (instead, only the definitions of affected situation predicates need to be changed, and this task is relatively trivial).

6.1.5 Adaptation layer

The adaptation layer provides support for the two programming models outlined in Chapter 5. Its principal responsibilities are twofold. First, it maintains common repositories of situation, preference and trigger definitions, which are shared by multiple applications. This requires the provision of facilities not only for applications to generate and retrieve these definitions, but also for users to browse and formulate their own preferences and triggers via user-friendly interfaces. The second responsibility of the adaptation layer is the evaluation of preferences and triggers on behalf of the application layer. A parser is invoked to translate the original textual forms into more flexible representations that support timely evaluation. For efficiency, the

(a)

HasChannel["Emma May", "emma@email.org"]*HasChannel*["Emma May", *channel*]*PersonLocatedAt*["Emma May", "98-122", -]

(b)

SynchronousMode("emma@email.org") \wedge *CanUseChannel*("Emma May", "emma@email.org")*possibly*(*ColleagueCall*(*caller*, *callee*)) \wedge \neg *WorkingHours*() \wedge \neg *Urgent*(*priority*)

(c)

EnterFalse(*Occupied*("Emma May"))*TrueToFalse*(*LocatedAt*("Emma May", "45-234") |*(TrueToPossibly*(*LocatedAt*("Emma May", "45-234"))) \rightarrow *PossiblyToFalse*(*LocatedAt*("Emma May", "45-234")))

Figure 6.2: Example queries. (a) Fact queries. (b) Situation queries. (c) Event queries.

parsing step is performed only once, upon the addition of a new definition, and the resulting representation is stored in preference to the textual form.

In order to evaluate a trigger or preference, the adaptation layer generates appropriate queries to the context repository using the query layer. As there is frequently an overlap between definitions (e.g., a single situation predicate recurs many times, as in the example preferences of Figure 5.5), and sets of similar evaluations often take place concurrently (e.g., when a preference is evaluated with respect to several candidate choices in order to generate a ranking of these), there is considerable scope and necessity for optimisation. The possibly large numbers of preferences and triggers that may be present in complex systems places further importance on efficient evaluation. A range of solutions can be employed, such as eager partial evaluation (involving the evaluation ahead-of-time of those components of situation predicates that are not context-dependent) and caching of full or partial results in order to eliminate redundant computations (and especially redundant invocations of the query interface).

Although logically a single component that maintains shared common repositories, the adaptation layer is in reality distributed to support efficient evaluation of preferences and triggers, as well as high availability, even in the face of disconnections. The distribution of the preference, trigger and situation information is managed such that information is situated with the applications that require it. As updates are relatively infrequent, and perfect consistency is not crucial, the replication of this information across the multiple locations at which it is required is desirable and not problematic.

As described previously in Section 5.5, the sharing, reuse and composition of situation and preference definitions is crucial in terms of usability and scalability; however, unlimited sharing is not always appropriate. Privacy concerns arise with the sharing of information (particularly preferences) in pervasive systems. This introduces an additional responsibility for the adaptation layer related to the maintenance and enforcement of access control policies. Various different policy types are required: for example, default preferences created by application developers can be granted unlimited visibility, organisational preferences should typically be visible to all of the members of the organisation, while preferences of individual users may be made available only to their owners or to trusted groups.

6.1.6 Application layer

The application layer provides further support for the branching and triggering programming models, and for formulating queries to the query layer. Broadly, it offers a programming toolkit that application developers can use to exploit the functions of the architecture within their programs. The toolkit provides (i) wrapper

classes that encapsulate functions of lower layers (such as `Context` and `Preferences` classes), (ii) special `Branching` and `Triggering` classes that implement the two programming models, and (iii) a variety of support classes such as `Valuation` and `Choice`. These are summarised in Table 6.1.

The core functionality of the toolkit is partitioned between the `Context`, `Branching` and `Triggering` classes. The first provides a wrapper for an instance of a context model maintained by the context management layer, and provides applications with access to the functions of the query layer. Instances of this class are also supplied as parameters to the methods of the branching and triggering toolkits, where they are used in the evaluation of preferences and triggers. The remaining two classes implement the programming models. The interface of the `Branching` class was introduced previously in Figure 5.9. The methods provided by this class are concerned with the scoring and ranking of choices against a specified context and preference, and selection and invocation of choices for which the scores meet certain criteria (such as exceeding a specified threshold). The `Triggering` class provides an implementation of the triggering model. Specifically, it offers the necessary tools for dynamically adding new triggers to the repository maintained by the adaptation layer, and for activating and deactivating existing triggers.

6.1.7 Summary

The roles of each of the six layers of the architecture are summarised in Table 6.2. The following section presents an analysis of the architecture and its underlying conceptual foundations with respect to the key requirements of pervasive systems.

6.2 Discussion

As motivated in Section 1.1, software for pervasive systems must satisfy the following requirements:

- support for *autonomy*, such that reliance on user input is minimised without materially diminishing user control;
- support for *dynamic computing environments*, such that software opportunistically exploits changing sets of computing resources;
- support for *dynamic user requirements*, such that applications are responsive to evolving goals and preferences;
- support for *scalability*, such that large volumes of information (including context, preferences and triggers) and large numbers of applications and devices (including sensors) can be accommodated; and

Table 6.1: Components of the programming toolkit.

<i>Class</i>	<i>Function</i>
Context	A wrapper for a context model maintained by the context management system. Represents a context against which preferences and triggers can be evaluated, and which can be queried using the situation, fact and trigger event abstractions.
Trigger	Represents a trigger managed by the adaptation layer. Triggers are managed using the Triggering toolkit (see below).
Preference, PreferenceSet	Represent a preference and preference set, respectively (as defined in Chapter 5). Preferences are supplied to the Branching toolkit (see below) to support rating, selection and invocation of choices.
Preferences	A wrapper for a repository of preferences maintained by the adaptation layer. Supports lookup of preferences and preference sets by name.
Situations	A wrapper for a repository of situation predicates maintained by the adaptation layer. Supports lookup by name. Situations can be referenced by queries on Context objects, and by preference and trigger definitions.
Branching	Provides a variety of methods that implement the branching model. These are listed in Figure 5.9.
Triggering	Provides methods that support the triggering programming model, allowing triggers to be created, activated and deactivated dynamically.
Valuation	Support class, used in the evaluation of preferences by the Branching toolkit, which implements a binding of variables to values.
Choice	Represents a candidate choice to be rated by the Branching toolkit. Each choice has its own Valuation and Action .
Action	Encapsulates the code to be invoked upon selection of the corresponding choice by the Branching toolkit.
Score	Represents a score assigned to a Choice by the Branching toolkit. This is either a numerical score in the range $[0,1]$ or one of the special scores.
Scores	A mapping from Choice instances to Score instances.

Table 6.2: A summary of the functions of the six layers of the architecture for context-aware systems.

<i>Layer</i>	<i>Function</i>
Context gathering layer	Responsible for acquisition of context information from sensors and processing, including data fusion and interpretation.
Context reception layer	Provides a bidirectional mapping between the context gathering and management layers. Input from the former is translated into the fact-based representation of the latter, and queries from the latter are routed to the appropriate components of the former. Also responsible for conflict resolution.
Context management layer	Manages a common repository of context information, and mappings and metadata for multiple context models. Enforces the constraints of these models, manages the special fact types, and implements relevant privacy policies.
Query layer	Provides applications and the adaptation layer with a convenient interface with which to query the context management system using the fact and situation abstractions.
Adaptation layer	Manages common repositories of situation, preference and trigger definitions, and evaluates these using the services of the query layer. Supports user-friendly browsing and editing of repositories and access control mechanisms.
Application layer	Provides a programming toolkit that affords applications with access to the query layer and offers support for the branching and triggering programming models.

- support for *resource limitations*, particularly in terms of battery power and network bandwidth in the case of sensing devices, and in terms of I/O capabilities in the case of user devices.

The following sections characterise the design features of the architecture that address each of these requirements.

6.2.1 Autonomy

As described in previous chapters (see, for example, Sections 1.1 and 5.1), the provision of an appropriate balance of software autonomy to user control is one of the key design challenges in pervasive systems. Context-awareness is seen as a solution to this challenge; however, the current generation of context-aware applications falls short of user expectations [44, 115, 116]. This is largely due to the fact that context-dependent behaviours are often:

- designed by application developers with little or no user consultation;
- static in nature, and incorporated directly into the application source code; and
- hidden from users, and difficult or impossible for users to override when the behaviours are inappropriate.

The solution offered by the architecture and programming models is to decouple context-aware functionality from the remainder of the application logic (in the form of triggers and preferences), such that this functionality is completely transparent and easily modified, even by unsophisticated users. Users can either specify preferences and triggers explicitly, or have these tailored to their requirements using automated elicitation techniques such as those described in Section 5.4.5. This allows applications to carry out actions on behalf of users in a highly autonomous fashion, with control ultimately lying with the user.

By itself, however, this solution may not be completely adequate, and the design of individual applications should exploit further opportunities for providing transparency and enabling users to control or override decisions when desired. A discussion of how this can be achieved in relation to an example application appears in the following chapter, in Section 7.4.1.

6.2.2 Dynamic computing environments

The support of the architecture for dynamic computing environments is twofold. First, adaptation of applications to changing environments is supported by treating resource characteristics as a type of context information. Second, the architecture

itself supports evolution (particularly of resources within the context gathering infrastructure) through the use of loose coupling between the architectural layers. The remainder of this section describes these two solutions in more detail.

Context-awareness has been widely exploited to allow applications to make opportunistic use of changing sets of computing resources. Scenarios that have been considered in this area include the use of context information to identify suitable printing services [33], and to adapt applications' communication patterns according to the available bandwidth and battery power [75]. These types of behaviour are easily supported within the framework of the proposed architecture, and the case study presented in the following chapter demonstrates concretely how the architecture and underlying models can be used to enable a communication application to adapt its choice of communication channel according to the devices available to users, by incorporating device information into the context model. This approach can be extended to accommodate other types of resource, such as input and output devices (allowing user interface adaptation) and network resources (allowing adaptation to network quality of service changes).

Within the architecture itself, dynamic changes in the resources of the context gathering layer (caused, for example, by the addition or removal of sensors, or migration between environments that are instrumented to differing degrees) are supported in two separate but complementary ways. First, the loose coupling in the lower layers, accomplished largely through the use of content-based routing to avoid explicit connections between components, allows context sensing and processing components to be added, removed and modified on-the-fly. Second, the separation of applications from their respective context models allows the latter to be evolved in response to changes in the context gathering infrastructure with minimal impact. This can typically be achieved with only corresponding changes to the affected situation definitions, and possibly also to some of the user preferences that are dependent on these. These are both minor tasks in comparison to modifying source code.

6.2.3 Dynamic user requirements

The architecture provides an unprecedented degree of support for customisation, as well as for user preferences that change over time, when compared to other architectures and frameworks for context-aware systems. In other solutions, dynamic application behaviour is achieved through adaptation to changes in context. In contrast, the architecture described here supports change along two dimensions: context, and user requirements with respect to context. This greater degree of flexibility is achieved by capturing user requirements explicitly in the form of preferences and triggers, which are divorced from the implementation and easily modified on-the-fly.

When applications take advantage of the branching and triggering models, a

wide range of behaviours can be achieved simply by editing the set of user preferences and triggers. This is demonstrated concretely in the following chapter, in which the influence of user preferences on the selection of communication channels for interactions between users is explored. Radical behaviour changes that require modification of the underlying context model are also possible, often without any modification of source code.

Further potential for highly adaptive behaviour is afforded by the ability to use learning techniques in order to elicit and evolve user preferences, as discussed in Section 5.4.5.

6.2.4 Scalability

Although not one of the central concerns of this thesis, the requirement for scalability is inherent in pervasive systems. Consequently, scalability was factored into the design of the architecture in several ways. These were already discussed intermittently throughout Sections 6.1.1 to 6.1.6, but are also briefly summarised here.

At the context gathering layer, the consequences of large numbers of components and frequent (and often continuous) sensor output are addressed through the use of selective forwarding of events using a content-based routing scheme, careful placement of processing components to enable data fusion and reduction at an early stage, and the use of the active update model for types of context for which the frequency of updates far outweighs the frequency of queries.

Within the context management layer, the storage and processing overheads associated with managing a separate repository of context information for each distinct context model are avoided by opting instead for a shared pool of data and a set of mappings from this pool to each of the models. This approach is combined with the use of selective distribution and replication in order to prevent bottlenecks and promote failure-tolerance, as well as strategies of active and real-time databases in relation to frequently changing data.

Within the query and adaptation layers, scalability is dependent on the ability to efficiently evaluate situation expressions, preferences and triggers. As described in Section 5.2.6, the situation abstraction was designed explicitly with efficient evaluation in mind. A variety of strategies, such as caching of results and eager partial evaluation, are also employed by the query and adaptation layers in order to reduce the evaluation time. Finally, the appropriate distribution of the adaptation layer enables situation, preference and trigger information to be situated with the specific applications that require it.

6.2.5 Resource limitations

Pervasive computing environments are characterised by extreme heterogeneity of computing devices and networking infrastructure. This implies that resource limitations must be tolerated and overcome in some areas of the architecture. This is particularly true at the two extremities (i.e., within the context gathering and application layers). In this section, a justification of the architecture's ability to accommodate various types of resource-poorness within these layers is given.

In general, the severest resource limitations are present within the sensing infrastructure. For most sensors, and particularly those that are mobile, the key factors are battery power and power-hungry resources, including network bandwidth and CPU speed/cycles. Correspondingly, the architecture places minimal requirements on these resources. A lightweight event-based communication model is used to transmit sensor data; however, in the case that the event generation software cannot be accommodated on the sensing device, the sensor data can be transmitted using an arbitrary form of communication to a proxy that produces the events on the sensor's behalf. Similarly, processing of the sensor data may or may not be carried out onboard the sensor, depending on the available resources. A trade-off between CPU and bandwidth requirements is often necessary (i.e., increased processing onboard the sensor can reduce bandwidth consumption, and vice versa).

Resource constraints with respect to the client devices on which user applications execute are generally less severe and of a different nature. Here, heterogeneity is often most noticeable with respect to input and output devices, owing to diverse contexts of use. For example, in-vehicle devices may rely mainly on speech-based input and output, while PDAs provide screen- and pen-based interaction. As discussed in Section 6.2.2, context-awareness offers a natural way to overcome these resource variations.

Bandwidth limitations and disconnections, as well as constraints on primary and secondary memory capacity, may also be problematic for some client devices. When this is the case, the distribution of functionality between the client device and other non-resource-constrained devices can be examined. Memory requirements can be minimised by offloading all of the functionality of the query and adaptation layers from the client device; the cost, however, is a diminished ability to handle disconnection. Bandwidth requirements can also be reduced by this solution, as communication between the application layer and the query and adaptation layers is typically less frequent than that between the latter layers and the context management layer; this is because each invocation of the query and adaptation layers can trigger multiple queries on the context repository.

The programming toolkit itself is lightweight, and places minimal requirements on the client device. However, when resource restrictions are severe, reduced-

functionality variants of the toolkit (for example, implementing the branching programming model without the triggering model) can be deployed.

6.3 An implementation of the architecture

A lightweight implementation of the architecture was produced, both as a form of validation for the design, and as a basis for practical experimentation with the novel context modelling approaches, preference model, and branching model developed in this thesis. With the latter goal in mind, the primary focus of the implementation was placed on the branching and query functionality of the application and query layers, the situation and preference management functionality of the adaptation layer, and the core management tasks of the context management layer. The functionalities of the query and adaptation layers were merged into the application layer toolkit for the purposes of the prototype, while those of the context gathering and reception layers were not implemented at all, as the development of sensing infrastructure falls outside the scope of this thesis and has been addressed extensively by previous research (see Section 3.1).

The prototype leverages commonly available tools, including a relational database management system (DBMS) and parser generator, and places an emphasis on simplicity over efficiency. Issues such as the distribution of the context and preference management infrastructures are not considered, as these have no bearing on the success of the primary goal of experimenting with, and thereby evaluating, the programming model and toolkit, and the underlying context and preference modelling techniques.

6.3.1 Organisation

The organisation of the prototype is depicted in Figure 6.3. There are two principal components:

- a programming toolkit, implemented in Java, which provides the combined functionality of the application, adaptation and query layers; and
- a context database, which forms a lightweight implementation of the context management layer.

The former is divided into query, branching and data components; these roughly approximate the functionality of the query, application and adaptation layers, respectively, and are further decomposed into varying numbers of classes. The latter closely correspond to the programming toolkit classes that were described in Table 6.1, except that the classes concerned with the triggering model have been omitted, and the **Context**, **Preferences** and **Situations** classes, instead of acting as

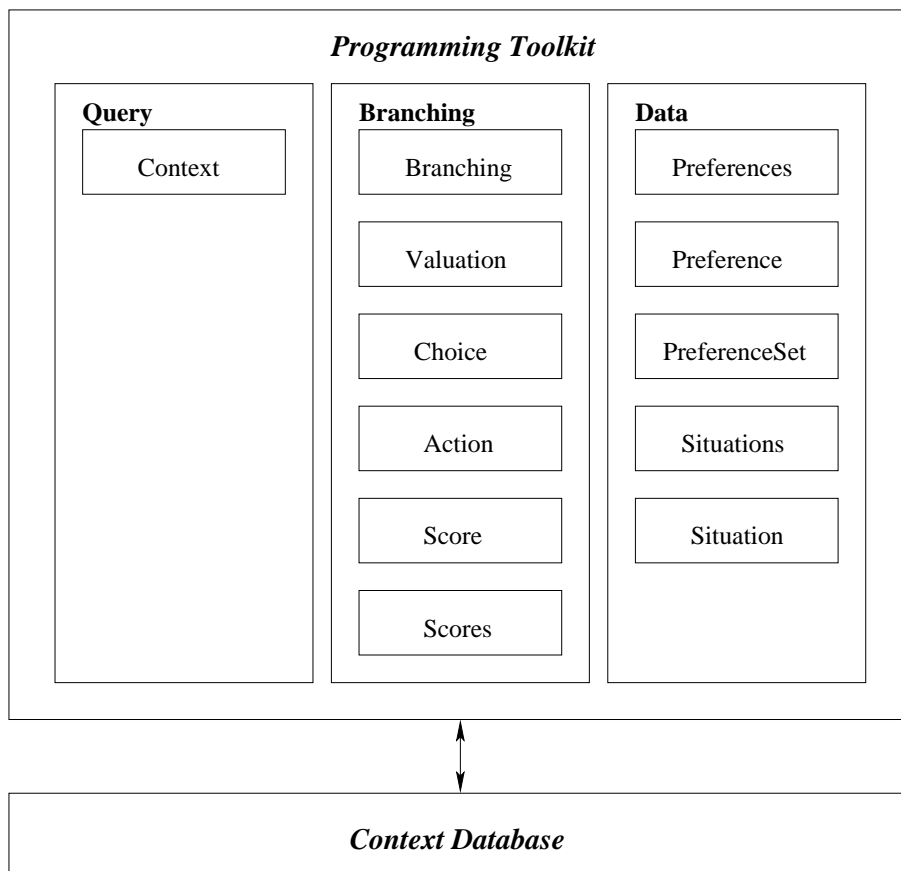


Figure 6.3: Organisation of the prototype.

wrappers for functionality of the query and adaptation layers, implement this functionality directly.

The following sections describe the implementation of each of the architectural layers as components of the prototype in further detail.

6.3.2 Implementation of the context management layer

As the primary focus of the prototype is on the branching programming model and supporting abstractions, and not on context management issues, only the core functions of the context management layer are implemented. The prototype exploits the relational mapping of the fact-based model, described in Section 4.6, by entrusting these functions to a conventional relational DBMS.

For the purposes of the prototype, the PostgreSQL DBMS [197,198] is used, and a variety of simplifying assumptions are made:

- the use of a common repository of data and separate mappings for each context model, as described in Section 6.1.3, can be easily realised with PostgreSQL using a single database with multiple corresponding views; however, as only a single context model is required for the purposes of the evaluation (see Chapter 7), the model is instead represented by a simple database;
- the distribution of context information amongst separate repositories is not considered;
- user-friendly interfaces that support manipulation of profiled fact types are not implemented; instead, this task is currently performed using the SQL-based client (psql) provided with PostgreSQL; and
- privacy is not supported in any form.

Moreover, the metadata and constraints associated with the context model are incompletely implemented. The remainder of this section describes the mapping of a selected subset of the metadata and constraints described in Section 4.6.8 to a representation within the relational database.

The realisation of basic constraints (primary key, key and inclusion dependency) are trivial in a database such as PostgreSQL. These are defined, using SQL commands, at the same time as the corresponding relations.

The *class* function, which maps each of the base relations to one of the values in the set $\{static, sensed, profiled\}$ is mapped to the special relation $Class(\underline{FactType}, Classification)$, in which the first attribute captures the name of a base relation and the second its classification. Similarly, the *altRole* function is captured by the relation $AltRole(\underline{FactType}, RoleName)$, where the two attributes represent the names of a relation and attribute/column within this relation, respectively.

The function *def*, which defines the derived relations in terms of base relations (using SQL statements), is not represented explicitly in the database, but is used to define an appropriate view for each derived relation.

Fact dependencies are not implemented, as these serve largely to support context management tasks which are not relevant to the goals of the evaluation.

6.3.3 Implementation of the query layer

As discussed in Section 6.3.1, the functionality of the query layer is incorporated into the **Context** class of the programming toolkit for the purposes of the prototype. This class implements fact and situation queries, as defined in Section 6.1.4, but omits event queries as these are principally used in relation to the triggering model, which is currently not supported.

The **Context** class takes advantage of the standard Java Database Connectivity (JDBC) API in order to perform queries on the relational database. This decouples the class from most of the implementation quirks of PostgreSQL, and allows another DBMS to be easily substituted in place of PostgreSQL if required. The JDBC API supports querying based on SQL; thus, the role of the **Context** class consists largely of translating queries into the appropriate SQL commands, dispatching these to the database using JDBC, interpreting the results according to the semantics of the fact and situation abstractions, and thereby deriving the result of the initial query.

The three types of supported query (logical fact, binding fact and situation) each require slightly different treatment, and are implemented by three separate methods of the **Context** class: **query**, **bind** and **evaluate**.

The **query** method is the simplest. This takes the name of a fact type and a list (array) of attribute values, and queries the database to determine whether a matching fact is present in the context repository. In accordance with the interpretation described in Section 4.6.9, the method returns *true* if there is a matching fact that has no alternatives, *possibly true* if there is at least one matching fact, and each fact either has alternatives or perfectly matches the query only when one or more nulls are replaced with arbitrary values, and *false* otherwise.

The **bind** method takes the name of a fact type and a mixed list of constant values and variables, and performs a matching process much like that of the **query** method. As a result of this process, a new binding of the variables to values is created for each matching fact that (i) has attribute values identical to each of the specified constant values (or else null values in place of these) and (ii) has no null values for any of the variable attributes.

Finally, the **evaluate** method takes a **Situation** instance and a valuation, and evaluates the former against the latter and the current context. The **Situation** is an already parsed situation predicate that contains an abstract syntax tree pro-

duced by the parsing process. (As the parsing of situations is the responsibility of the adaptation layer, this step will be described further in the following section.) The abstract syntax tree supports direct evaluation by invoking `evaluate` on its root node; this often leads to further fact and situation queries, depending on the definition of the situation.

6.3.4 Implementation of the adaptation layer

Within the prototype, the functionality of the adaptation layer is encapsulated in five classes of the programming toolkit, as shown in Figure 6.3. These classes support the creation and management of repositories of preferences, preference sets and situation predicates, and the dynamic evaluation of preferences and situations. The functionality of the adaptation layer in relation to triggers, as described in Section 6.1.5, is currently omitted from the prototype. Similarly, access control mechanisms and optimisations of the evaluation procedures are not implemented, and only primitive support for user editing of situation and preference information is provided. These restrictions simplify the implementation without hindering the goals of the evaluation.

The `Preferences` and `Situations` classes are responsible for managing repositories of preferences and preference sets, and situation predicates, respectively. These support operations such as insertion, deletion, and name-based lookup. The current method of initialising the repositories for a given application is straightforward, and involves the use of configuration files that specify the preference and situation information in textual form. The contents of the files are interpreted using a parser, specially constructed for the task using the JavaCC parser generator tool [199]. The result of the parsing step is a compact abstract syntax tree (AST), which can be evaluated by calling the `evaluate` method on the root node.

Instances of the `Preference`, `Situation` and `PreferenceSet` classes represent named preferences, situation predicates and preference sets, respectively. The first two classes act as wrappers for the ASTs that result from the parsing process, while the latter maintains a list of `Preference` instances and supports operations such as lookup, insertion, deletion and iteration over preferences.

6.3.5 Implementation of the application layer

The remainder of the prototype implements the branching functionality of the programming toolkit, as it is described in Sections 5.4.4 and 6.1.6.

The `Branching` class forms the heart of this toolkit, and implements methods that are concerned with the scoring, ranking, selection and invocation of a set of choices with respect to a specified preference, context and valuation of variables. These methods were detailed earlier in Section 5.4.4 and Figure 5.9. The remain-

ing branching classes (shown in Figure 6.3) perform supporting roles as described previously in Table 6.1.

6.4 Discussion and future work

This chapter served to integrate the theoretical results of Chapters 4 and 5 in two ways. First, it combined the context modelling solutions, preference model and programming techniques into a layered architecture for context aware systems. This architecture offers a complete solution, from context acquisition through to application-level support for the programming models, and, as shown in Section 6.2, addresses the key requirements of pervasive systems. Second, as a form of proof-of-concept, the chapter introduced a lightweight prototype of the architecture in the form of a programming toolkit and supporting context database. By leveraging commonly available tools where possible, the toolkit implements selected core aspects of the architecture in approximately 6,000 lines of Java code, including all of the principal functions of the application, adaptation and query layers required by the branching programming model. The prototype serves as an important testbed for the key research contributions of this thesis, and forms the basis for the case study that appears in the following chapter. This case study explores the use of the toolkit (and its theoretical underpinnings) as a platform for the development of a context-aware communication application, yielding a variety of practical results.

As the prototype represents an incomplete mapping of the architecture, there is considerable scope for extension in the future. The main omission in terms of the application, adaptation and query layers is support for the triggering model. However, there is, additionally, considerable scope for optimisation within the implementations of the adaptation and query layers, particularly in terms of the evaluation of preferences and situation predicates.

The context management layer has been realised in only a skeleton form, leaving open many areas for future work. Amongst these are the special management tasks associated with fact dependencies and temporal and sensed fact types (including support for the active update model), as well as the distribution and replication of context information.

The context gathering and reception layers are not yet present in the prototype in any form. The context information used to populate sensed fact types is instead produced by simulation. A variety of suitable solutions for the lower layer, described in the survey of Section 3.1, have previously been implemented, and it would be instructive to integrate one of these with the prototype in order allow the real-time issues associated with live sensor data to be better evaluated.

Finally, the prototype does not address privacy at any of the levels. As complete privacy solutions are not yet available for pervasive systems, this represents a crucial

research problem as well as an implementation one.

Chapter 7

Case study: Context-aware communication

This chapter describes a case study involving the development of a prototypical context-aware application using the novel context and preference modelling approaches, and associated software infrastructure, introduced in the preceding three chapters. The case study revolves around the scenario concerned with context-aware communication that has been used as a running example throughout this thesis. The principal contributions of this chapter are twofold. First, the process and challenges associated with the design and implementation of context-aware software are illustrated by example, and, second, the principal research contributions of this thesis (i.e., the aforementioned modelling approaches and infrastructure) are evaluated and validated.

The structure of the chapter is as follows. Section 7.1 introduces the problem domain that is addressed by the communication application, and Section 7.2 outlines the key design objectives, deriving inspiration from related research in context-aware communication. Section 7.3 presents the results of an informal study that was performed in order to discover the key user requirements in relation to the application, and Section 7.4 addresses design challenges associated with autonomy and privacy. Section 7.5 briefly outlines the process by which the design of the application was developed, while Sections 7.6 to 7.10 present the results of the various design phases. Specifically, Section 7.6 outlines the core functionalities of the application, and Section 7.7 characterises the context-aware behaviour in more detail. Sections 7.8 and 7.9 explore the modelling of context and preferences in relation to the application. Section 7.10 concludes the discussion of the application design by outlining the key implementation components. Next, Section 7.11 describes the process and issues involved in implementing the design, while Sections 7.12 and 7.13 conclude the chapter with a discussion of the results of the case study and issues for future research.

7.1 Introduction

As described in Chapter 1, context-aware communication has been explored in various forms in the past. Many of the applications that have been developed in this domain address niche areas, such as telephony [7, 23, 42, 43] or text-based messaging [8–10, 45, 46]. The goal of the application explored here is broader. The application takes the form of an integrated communication platform, in which the choice of communication channels for interactions between users is mediated by context-aware agents. This context-based choice problem forms an interesting testbed for the novel branching model introduced in Chapter 5, particularly as there is considerable scope for exploiting diverse sets of user preference information as a basis for channel selection.

The application, named *Mercury* after the messenger in Roman mythology, operates as follows. Each agent manages (and maintains a centralised record of) the interactions of a given user over a variety of communication channels, including telephone, email, text messaging and videoconferencing. Sequences of related interactions are organised into threads, termed *dialogues*.

When a user requires an interaction with one or more other people, the communication agent is invoked. The agent, in cooperation with the agents of the other participants, consults the contexts and preferences of each party in order to recommend an appropriate channel for the interaction. The usage scenario that was presented first in Section 1.5 and again in 5.2.4 illustrates some of the factors that are taken into consideration, amongst which are user activity, priority, relationships between the participants and available communication devices.

7.2 Objectives

The design goals of the application can be characterised with respect to the five objectives identified by Schilit et al. [44] for context-aware communication. These are:

1. improving relevance of communications to context;
2. minimising disruption;
3. improving awareness by supplying users with information that will lead to intelligent decisions;
4. reducing overload by filtering irrelevant communications; and
5. selecting channels appropriate to both the caller and the callee.

Mercury addresses all of these objectives, with particular focus on the second and last. Disruption is minimised not only by taking user activity into account, but also by allowing users to specify explicitly (through their preferences) which forms of interruption are acceptable under which circumstances. For instance, users can choose to only admit high-priority telephone calls during meetings. This selective form of screening is difficult (if not impossible) to accomplish using traditional forms of communication.

The selection of communication channels also respects other classes of user preference (such as the general preference of a user for email over telephone for non-urgent interactions), and exploits context information to avoid inappropriate channels (such as videoconferencing, when the required equipment is not available).

In order to minimise irrelevant communications in accordance with the first and fourth objectives, Mercury applies an expiry scheme to automatically delete outdated interactions. Each interaction is tagged with an expiration date that designates its useful lifetime; this allows messages such as seminar announcements to be removed once their usefulness has passed (in this case, this would probably occur at the time of the seminar).

While Mercury does not allow users to reject interactions from others outright, the nuisance factor associated with unsolicited communications is minimised by the ability to route these, using appropriate preferences, to innocuous channels (for example, an email account that is reserved for spam and checked infrequently).

The remaining objective, of providing users with increased awareness, is addressed indirectly. The user is not presented personally with information about the context of others, as this design opens up considerable potential for privacy violations. Each user's agent, however, maintains this awareness and proposes actions that are consistent with the context.

7.3 Requirements analysis

Prior to the design of the application, a small, informal study was conducted in order to investigate patterns of interpersonal communication, as well as the factors involved in the choice of communication channels for interactions. The study took place over a period of two days, in a university research setting, and involved the recording of all interactions for the individuals participating in the study (including incoming and outgoing telephone calls, email sent and received, and so on). The justifications that the participants gave for their choices of communication channels were also recorded.

On the basis of this study, all of the interactions were classified as belonging to one of three distinct types of dialogue:

- *queries*, in which the initiator of the dialogue directed a brief enquiry to one or more recipients, which was followed up by zero or more replies;
- *discussions*, which involved a more protracted and less orderly sequence of interactions between two or more parties; and
- *announcements*, in which the initiator communicated some information to one or more recipients without the expectation of a response.

A user's choice of communication channel was found to be largely dependent on:

- the type of dialogue (e.g., announcements were almost always made using unidirectional channels, such as email, while bidirectional channels were more common for queries and discussions);
- the number of involved parties (e.g., once more than two parties were involved, email became considerably more common for discussions and queries than the telephone);
- priority (e.g., synchronous channels were favoured for important and time-critical interactions);
- the perceived activities of other users (e.g., participants sometimes quickly responded to a new email with a telephone call or even a visit to a colleague's office, feeling confident of finding the sender of the email still available); and
- generic user-specific preferences (e.g., some users stated a general preference for using email over the telephone, which appeared to be independent of other factors, such as the type of dialogue).

The design of the communication application, which is the subject of the following sections, was driven largely by these findings.

7.4 Design considerations

As discussed in previous chapters (see, for example, Sections 1.1 and 5.1), context-aware applications are subject to a variety of novel design constraints. In this section, the issues of autonomy and privacy are examined, and the design features that are introduced to address these are briefly outlined.

7.4.1 Autonomy and control

A key challenge associated with the design of an agent-based application such as Mercury lies in balancing the autonomy of the application against a sufficient degree of user control. This problem has already been discussed in a general sense in

Sections 1.1, 5.1 and 6.2.1. When considering the domain of context-aware communication, the challenge translates to apportioning appropriate degrees of control to the initiator of an interaction, the recipient(s), and the communication agents. In traditional communication systems, control lies largely with the initiator. As argued by Wams and Van Steen [45], there is great benefit in transferring a large share of this to the recipient(s).

There is additionally a strong justification for retaining humans in the decision loop. Schilit et al. [44] survey a variety of communication applications, and classify these with respect to their autonomy in terms of both acquisition of context information and action upon context information. They demonstrate that high autonomy in both dimensions is typically undesirable, as it removes human common sense.

Accordingly, Mercury approaches the control problem as follows. Selection of communication channels is carried out through a negotiation process that consults the preferences of all involved parties. The requirements of the recipient(s) are consulted in the first instance. Each recipient's agent supplies the initiator's agent with a list of suitable communication channels formed according to the recipient's preferences and context¹. The initiator's agent next uses the initiator's preferences to rank the resulting list according to desirability, and to cull inappropriate choices. A set of the highest ranked channels (or the complete set of channels when only one or two remain) is presented to the initiator, who makes the final choice (or cancels the interaction if desired).

7.4.2 Privacy

The privacy issues associated with uninhibited collection and sharing of context information are widely recognised, and are believed to represent one of the most significant barriers to the adoption of pervasive computing applications [171, 200]. Moreover, the explicit modelling of user requirements, in the form of preferences and triggers, as described in the previous chapters, introduces further challenges with respect to privacy. A study of users' privacy expectations with respect to their preference and trigger information is clearly beyond the scope of this research; however, it is easy to imagine scenarios in which unrestricted sharing is undesirable. For example, in the case that users create preferences assigning low priority to communications involving particular individuals (perhaps in order to filter unwanted calls and messages), they would typically not want to reveal these preferences to others.

As described in Chapter 6, the management and enforcement of privacy policies must be largely the responsibility of the context and preference management infras-

¹Direct consultation with the recipient at this stage is generally intrusive and undesirable, but may be performed in some cases if permitted by the recipient's preferences

structure. However, privacy requirements must also be factored into the design of context-aware applications. The design of Mercury specifically must ensure that the objectives outlined in Section 7.2 are met (i.e., user disruption is minimised, missed calls are avoided, and so on) without exposing sensitive types of user context or preference information.

The solution adopted by Mercury is straightforward: the context and preferences of each user are evaluated only by the user's own agent. When selecting a communication channel for an interaction, a set of communication agents arrive at a suitable choice through a negotiation process in which each user's agent reveals only the channels that are suitable for the user (and not the context and preference information that were used to derive these). Although an argument could be made for improving the final choice of channel by revealing selected aspects of the recipients' contexts to the initiator of the interaction, this approach is fraught with dangers, as it is not clear which types of context should be regarded as safe to expose (and these would generally change depending on the users involved, and the nature of their relationships with one another).

7.5 Design process

The next part of this chapter presents a design that incorporates the functionality described in Section 7.1, and additionally satisfies the objectives and requirements introduced in Sections 7.2 to 7.4. This design was derived via the following steps:

1. identification of the core application functionality;
2. design of context-aware behaviour;
3. context modelling;
4. preference modelling; and
5. component design.

Sections 7.6 to 7.10 present the respective outcomes of these five phases.

7.6 Core functionality

This section identifies Mercury's key objectives, focusing first on the core functionalities and their translations into the user interface design, and then refining the scope of the application in terms of the supported communication channels.



Figure 7.1: The main window of the Mercury user interface.

7.6.1 Overview

Broadly, Mercury's role is to manage the three classes of dialogue between users that were introduced in Section 7.3. Mercury is agent-based, with each user having a corresponding communication agent responsible for:

- managing a record of the dialogues which involve the user;
- managing a repository of contact information, somewhat resembling the address book of an email client; and
- assisting with the selection of communication channels for new interactions.

The third task is the most important, and will be the primary focus of the sections that follow.

The three core functionalities are reflected in the main window of the user interface design, shown in Figure 7.1. Arranged in a similar manner to an email client, the largest portion of the window depicts a history of dialogues. Each dialogue occupies a row, listing the number of the dialogue, its type, subject, and sequence of interactions. For each interaction, the time, initiator, channel (represented by the identifier of the initiator's endpoint) and priority are shown.

The main operations on the dialogue and contact lists are invoked using the toolbar (or, alternatively, menus or hotkeys). The three leftmost buttons on the toolbar allow the user to initiate a new query, discussion or announcement, respectively. The fourth button creates a new interaction within an existing dialogue, while the last enables the user to edit the list of contacts.

7.6.2 Scope

The types of communication channel supported by Mercury are not limited; however, the design focuses on the following:

- email;
- telephone;
- videoconferencing;
- text messaging using the Short Messaging Service (SMS); and
- text messaging using the Tickertape application.

The first four channel types require no explanation. Tickertape [201] is a lightweight awareness and communication tool, based on the Elvin notification service, which displays short text messages in a scrolling line. The tool requires minimal screen space, and is intended to run continuously on the desktop, remaining in the periphery

of the user's attention until a message appears. Messages automatically disappear after a fixed period, and can also be dismissed directly with a mouse click. Tickertape exploits the Elvin content-based routing scheme, such that messages are not routed to explicitly specified addresses, but instead are delivered in accordance with user subscriptions. The tool is deployed widely within the Distributed Systems Technology Centre at which it was developed, as well as in a range of other organisations, and is commonly used for interactions such as announcements (e.g., when a party is heading to lunch) and brief general queries (e.g., "has anybody seen x?").

As the principal goal of the case study is to investigate the task of selecting communication channels, the establishment of interactions using the chosen channels and subsequent context-driven adaptation (such as switching from full videoconferencing to streaming of voice only in response to a quality of service degradation) are not examined. These problems are the subject of a separate and ongoing research project.

7.7 Context-aware behaviour

Although the potential uses of context in communication are broad, as demonstrated by the wide variety of applications developed in this area, the use of context in Mercury is currently confined to the channel selection problem. The investigation of additional uses of context, such as user interface adaptation (e.g., to trigger switching between voice-based interaction, in the style of Nomadic Radio [123, 202], and the currently implemented GUI-based interaction) is left as an area for future research. This section formalises the negotiation protocol used to perform channel selection, and highlights the role of context and preferences.

The negotiation procedure is invoked by a user (the initiator) wishing to establish communication with one or more other users (the recipient(s)). The initiator negotiates the series of dialog boxes illustrated in Figures 7.2 to 7.5. When creating a new dialogue, the initiator is prompted for a subject and set of recipients, as shown in Figure 7.2. Next, the initiator specifies a priority and expiry time for the current interaction using the dialog box in Figure 7.3². As individual interactions within a dialogue may have different sets of participants (e.g., when new participants are involved mid-dialogue), the set of recipients can be edited at this stage.

Following these initial steps involving the user, the initiator's agent (IA) carries out a behind-the-scenes negotiation protocol with the recipients' agents (RA). The steps involved in this protocol are illustrated (for the case involving four parties) in Figure 7.6. The IA sends a channel request to each RA, specifying the name of the initiator, the type of dialogue required, the priority of the interaction, and

²Alternatively, when creating a new interaction in an existing dialogue, this is the first step.

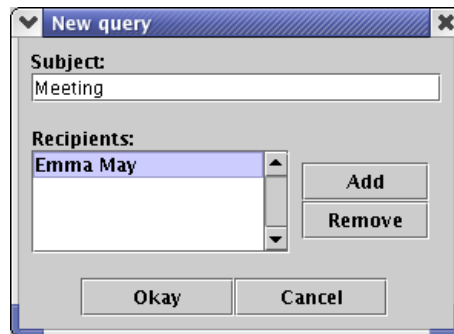


Figure 7.2: New query dialog.

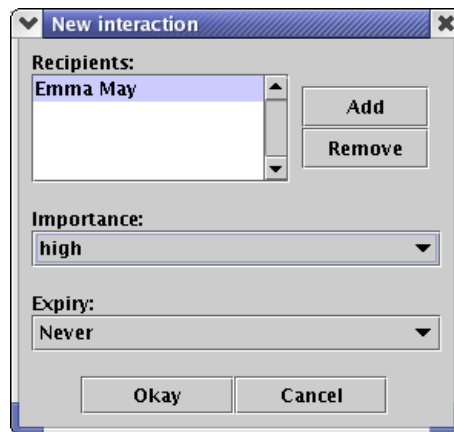


Figure 7.3: New interaction dialog.

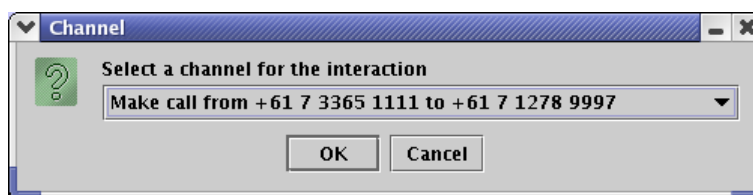


Figure 7.4: Channel selection dialog.

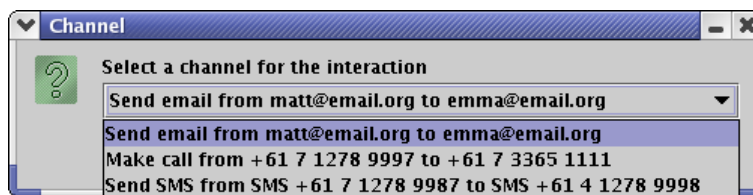
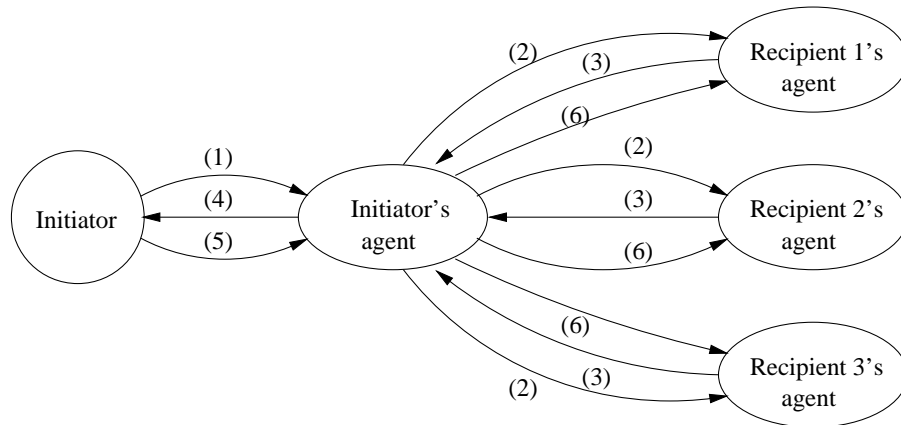


Figure 7.5: Channel selection dialog, showing a drop-down list of options.



- (1) Request for new interaction
- (2) Request for n channel proposals (parameters: n , initiator name, dialogue type, priority)
- (3) Proposed channels (tailored to recipient's context and preferences)
- (4) Proposed channels (merged from (3); culled and sorted using initiator's preferences)
- (5) Channel selection
- (6) Request for new interaction via selected channel

Figure 7.6: Channel negotiation protocol.

n , the desired number of channel proposals. Each RA responds with a list of up to n channels that are deemed suitable for the interaction, given the recipient's current context and preferences. The IA merges the proposals to derive a list of channels suitable for all parties, filters and sorts these according to the context and preferences of the initiator, and then presents the highest ranked proposals to the initiator for the final selection using the dialog box illustrated in Figures 7.4 and 7.5³. Finally, the IA sends a request to each RA to establish the interaction using the selected channel.

In some cases, particularly when there are many recipients, there may be no channels remaining after the preferences of all parties are accounted for. In this case, the parameter n may be increased and a second round of negotiation attempted, or the protocol may be terminated, and the choice of a further course of action delegated to the initiator.

In the two sections that follow, the task of modelling the context and preference information required to implement this negotiation process using the branching model is addressed.

7.8 Context modelling

Context and preference modelling are largely interdependent tasks, as the types of information included in the context model limit the forms of preference that can be

³Note that the initiator remains able to opt-out at this stage, if none of the proposed channels are desired.

expressed. Mercury's context and preference models were derived using an iterative process. That is, a basic context model was constructed, sample preferences were formulated against this model, the model was further refined using feedback from the second step, and, finally, the preferences were redefined against the new model.

An accurate understanding of user requirements is especially important to the success of both modelling tasks. Although the ability to add new preferences (and even augment the context model), as required, alleviates the obligation to perfectly capture user needs at the design phase, an early understanding of the key issues clearly leads to a more robust design. In light of this, the results of the informal requirements study described in Section 7.3 (and, in particular, the main factors cited by the study participants in relation to channel selections) were continually consulted at this stage of the design.

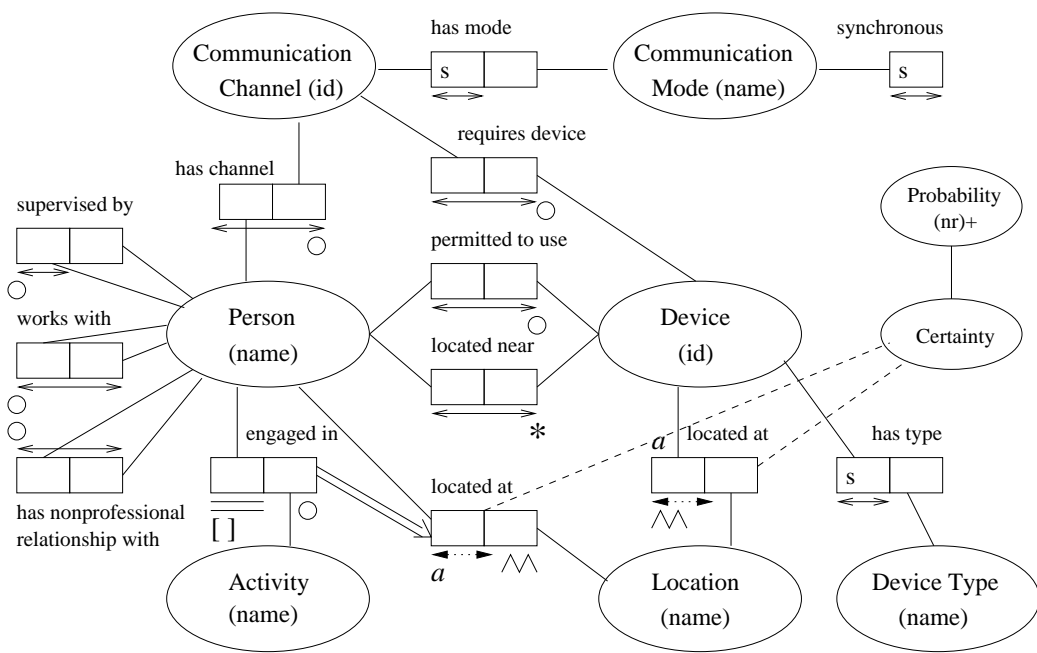
The context modelling process encompassed two steps: detailed model design using CML, and specification of a set of abstract contexts using the situation abstraction. The detailed design identified the required fact types. The following broad classes of information were included in the model:

- associations of people to communication channels and devices;
- activities of people;
- relationships between people, including supervisor, colleague and personal (i.e., non-professional) relationships;
- properties of communication channels and devices (including types of both);
and
- locations of people and communication devices (and proximity between these).

All of these classes of information are profiled or static, excepting the last (of which the location information is sensed, and proximity information is derived). Of the profiled information, only user activity is highly dynamic; this is extracted from users' schedule information, while the other types of profiled information, and all of the static types, are specified directly by users.

In accordance with the CML modelling methodology presented in Chapter 4, temporal fact types, dependencies and alternative fact types were identified; and quality annotations were added as required. As the steps and considerations involved in this modelling process were enumerated in detail in Chapter 4, they will not be discussed further here. The resultant model, reproduced from Figure 5.1, is illustrated in Figure 7.7. The relational mapping is not shown, but can be seen in the original figure.

In this case study, the information used to populate the context model is completely simulated (that is, no live sensor data is used). Were this not the case, a



* located near(p,d) iff located at(p, l1)
and located at (d, l2)
and l1 = l2

engaged in(p1,a) dependsOn located at(p2,l)
iff p1 = p2

Figure 7.7: The detailed context model.

further part of the design process would involve identifying suitable sensing infrastructure that could be used to populate the two sensed fact types.

The task of modelling abstract situations in terms of the fact-based context model is closely aligned with the preference modelling task. The goal is to define a core set of situations that can be easily combined to define preference scopes. This set should be concise but comprehensive (bearing in mind that users can define new situations to cover unusual requirements, but should not need to do so often), and each situation should be narrow in scope so as to form a suitable building block for later composition.

The set of situation predicates that was designed for use with Mercury is shown in Figure 7.8; this set represents a superset of the example predicates already presented in Chapter 5. The reader will note that a slightly different format is used in the figure compared to that used in Chapter 5. This new format is the one implemented by the situation parser, which was described briefly in Section 6.3.4.

7.9 Preference modelling

There are two principal design tasks related to preferences. The first is a general specification of the classes of preference that are required by the application, while the second is the design of a default preference set which captures generic user requirements, and which can be used unmodified if required, or extended to include user-specific preferences. The second task is typically not restricted to the design phase, but instead is revisited when the implementation of the application is complete, and empirical evaluation and fine-tuning are possible.

Mercury's channel selection protocol requires two types of preference for each user: one for the case involving the user as the initiator of an interaction, and the other for the case in which the user participates as a recipient. These have slightly different uses. The recipient preferences are principally aimed at minimising unwanted interruptions and preventing the use of inappropriate channels. In addition to parameters such as the priority of the interaction and the dialogue type, these take into consideration the recipient's context, including the current activity and the set of available communication channels (determined by the set of resources in close proximity to the user). The initiator preferences have a lesser role, as the initiator is directly involved in the channel negotiation process and is able to represent his/her interests. These preferences are exploited to cull any obviously inappropriate channels from the proposals made by the recipient(s), and to sort the remaining channels by preference prior to presentation to the initiator for the final selection.

The default preferences that were designed for Mercury are shown in Figures 7.9 and 7.10. (Again, the format matches that implemented by the adaptation layer's parser, and thus deviates slightly from the format of Chapter 5.)

```

SupervisorCall(caller, callee) : SupervisedBy[caller, callee];
ColleagueCall(caller, callee) : WorksWith[caller, callee] OR
    WorksWith[callee, caller];
PersonalCall(caller, callee) :
    NonProfessionalRelationship[caller, callee] OR
    NonProfessionalRelationship[callee, caller];
WorkingHours() : workinghours(timenow()) = TRUE;
LocatedAt(person, place) : EXISTS probability .
    PersonLocatedAt[person, place, probability] .
    probability > 0.8;
Occupied(person) : EXISTS t1, t2, activity .
    EngagedIn[person, activity, t1, t2] .
    (t1 <= timenow() AND timenow() < t2 AND
    (activity="meeting" OR activity="taking call"));
CanUseChannel(person, channel) : FORALL device .
    RequiresDevice[channel, device] .
    (LocatedNear[person, device, _] AND
    PermittedToUse[person, device]);
SynchronousMode(channel) : FORALL mode .
    HasMode[channel, mode] .
    Synchronous[mode];
Urgent(priority) : priority = "high";
IsQuery(dialoguetype) : dialoguetype = "query";
IsAnnouncement(dialoguetype) : dialoguetype = "announcement";
IsDiscussion(dialoguetype) : dialoguetype = "discussion";
IsEmail(channel) : EXISTS mode .
    HasMode[channel, mode] .
    mode = "email";
IsTelephone(channel) : EXISTS mode .
    HasMode[channel, mode] .
    mode = "telephone";
IsSMS(channel) : EXISTS mode .
    HasMode[channel, mode] .
    mode = "SMS";
IsTickertape(channel) : EXISTS mode .
    HasMode[channel, mode] .
    mode = "tickertape";
IsVideoconference(channel) : EXISTS mode .
    HasMode[channel, mode] .
    mode = "videoconference";
MultipleRecipients(recipients) : recipients > 1

```

Figure 7.8: Mercury's core set of situation predicates.

```

DefaultRecipientPrefs = {
p1 =  WHEN SynchronousMode(channel) AND Occupied(recipient)
      AND NOT Urgent(priority)
      RATE prohibit,
p2 =  WHEN Urgent(priority) AND SynchronousMode(channel)
      RATE 1,
p3 =  WHEN Urgent(priority) AND NOT SynchronousMode(channel)
      RATE 0.3,
p4 =  WHEN NOT Urgent(priority)
      RATE 1
};
DefaultInitiatorPrefs = {
p5 =  WHEN IsQuery(dialoguetype) AND IsEmail(channel)
      RATE 1,
p6 =  WHEN IsQuery(dialoguetype) AND IsTelephone(channel)
      RATE 1,
p7 =  WHEN IsQuery(dialoguetype) AND IsSMS(channel)
      RATE 0.7,
p8 =  WHEN IsQuery(dialoguetype) AND IsTickertape(channel)
      RATE 0.8,
p9 =  WHEN IsQuery(dialoguetype) AND IsVideoconference(channel)
      RATE 0.3,
p10 = WHEN IsDiscussion(dialoguetype) AND IsEmail(channel)
      RATE 0.8,
p11 = WHEN IsDiscussion(dialoguetype) AND IsTelephone(channel)
      RATE 1,
p12 = WHEN IsDiscussion(dialoguetype) AND IsSMS(channel)
      RATE 0.2,
p13 = WHEN IsDiscussion(dialoguetype) AND IsTickertape(channel)
      RATE 0.5,
p14 = WHEN IsDiscussion(dialoguetype) AND
      IsVideoconference(channel)
      RATE 0.8,
p15 = WHEN IsAnnouncement(dialoguetype) AND IsEmail(channel)
      RATE 1,
p16 = WHEN IsAnnouncement(dialoguetype) AND IsTelephone(channel)
      RATE 0.2,
p17 = WHEN IsAnnouncement(dialoguetype) AND IsSMS(channel)
      RATE 0.5,
p18 = WHEN IsAnnouncement(dialoguetype) AND IsTickertape(channel)
      RATE 0.9,
p19 = WHEN IsAnnouncement(dialoguetype) AND
      IsVideoconference(channel)
      RATE 0.2,
};

```

Figure 7.9: Mercury's default initiator and recipient preferences.

```
GenericPrefs = {
p20 = WHEN SynchronousMode(channel) AND
        NOT CanUseChannel(user, channel)
        RATE prohibit,
};

CombinedRecipientPrefs =
    WHEN true
    RATE average(DefaultRecipientPrefs);

CombinedInitiatorPrefs =
    WHEN true
    RATE average(DefaultInitiatorPrefs);

CombinedGenericPrefs =
    WHEN true
    RATE average(GenericPrefs);

RecipientPrefs =
    WHEN true
    RATE average(<CombinedRecipientPrefs,
                CombinedGenericPrefs>);

InitiatorPrefs =
    WHEN true
    RATE average(<CombinedInitiatorPrefs,
                CombinedGenericPrefs>);
```

Figure 7.10: Mercury's generic and composite preferences.

The recipient preferences, shown in Figure 7.9, are designed to capture a minimal set of requirements that are virtually universal. The first preference prohibits the use of synchronous modes of communication (such as telephone and videoconference) when the recipient is engaged in a task that should not be interrupted⁴. The second and third preferences together ensure that synchronous modes of communication are rated more highly than asynchronous modes for urgent interactions. The final preference is present only to ensure that all candidate choices receive a non-indifferent score; this can be replaced by the user with more specific preferences for non-urgent interactions, if desired.

The initiator preferences are concerned with ranking channels according to their suitability for the current dialogue type. For example, they imply that email and telephone are the most appropriate channel types for queries, followed by Tickertape, SMS, and finally videoconferencing.

There is one generic preference, common to both initiator and recipients, shown in Figure 7.10. This prohibits the use of any synchronous communication channel for which the user lacks access to the required device(s).

As shown in Figure 7.10, the combination of the preferences into overarching recipient and initiator preferences, suitable for use with the branching model, is straightforward and relies on the *average* function defined in Section 5.4.3.

7.10 High-level design

This section concludes the discussion of the application design by sketching an outline of the implementation components. As this aspect of the design is much the same as for a traditional application that lacks context-aware features, its description is brief.

The functionality of each communication agent is partitioned between three components, as shown in Figure 7.11. The *User Interface* component implements the graphical user interface, which is composed largely of the windows and dialogs depicted in Figures 7.1 to 7.5. The *Data* component implements persistent stores of dialogue, contact and configuration information. The majority of the user interface functions directly trigger operations on these data stores. Finally, the *Negotiation* component implements the channel selection protocol that was described in detail in Section 7.7. This is initiated upon the creation of a new interaction within the Data component. In order to execute the protocol, a naming service is invoked to discover the communication agents of the recipients, and the set of exchanges between the agents, shown in Figure 7.6, is carried out using remote method invocations. At various stages, the Negotiation component exploits the services of the

⁴At present, this preference is invoked when the recipient is in a meeting or busy with another call; however, the definition of the *Occupied* predicate can be refined to encompass other activities.

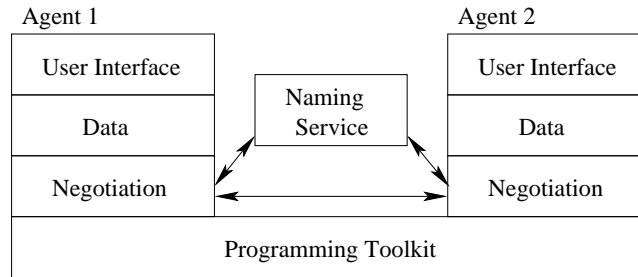


Figure 7.11: High level design of the prototype.

software infrastructure described in the previous chapter, using the programming toolkit. Specifically, it uses the query services of the toolkit to retrieve lists of candidate channels from the context repository, and the branching methods to perform channel selection and sorting.

7.11 Implementation

A prototypical implementation of Mercury was produced in Java [203], using Swing to generate the user interface, and Java Remote Method Invocation (RMI) to carry out the exchanges between agents. This section describes the main steps involved in the implementation process, with a focus on those aspects involving context, preferences and the programming toolkit.

The implementation involved three tasks:

- instantiation and population of the context model;
- population of the situation and preference repositories; and
- implementation of the User Interface, Data and Negotiation components described in the previous section.

In order to instantiate the context model, a mapping to the relational model was carried out as described in Section 4.6. The basic relations resulting from this procedure were presented earlier in Figure 5.1 (b). Additional metadata and constraints were mapped to the database as described in Section 6.3.2. The creation of a new database containing these relations (and appropriate populations of data) could have been performed either manually using the SQL client provided with the PostgreSQL database (psql), or via Java code using the JDBC database API to generate and execute the required SQL commands. The latter approach was adopted. The context initialisation code is straightforward, comprising approximately 500 lines of code (of which much is simply concerned with generating the appropriate SQL commands and performing exception handling, and is reusable for other context models

Table 7.1: Relative number of classes and lines of code for the three implementation components.

<i>Component</i>	<i>Combined number of classes and interfaces</i>	<i>Lines of code (approx.)</i>
User Interface	12	1300
Data	11	700
Negotiation	2	500

and applications)⁵. Further run-time context changes are carried out by hand using the psql tool to allow maximum flexibility. This allows the context to be changed arbitrarily for the purpose of experimentation.

As the prototype relies on simulated context information in place of live output from sensors, there was no necessity of programming the context receptors that map data from the context sensing infrastructure to the database repository. This would generally not be a demanding task, provided that infrastructure capable of producing information close to the required level of abstraction was already in place.

The population of the situation and preference repositories was trivial. As described in Section 6.3.4, situations and preferences are read from file by the adaptation layer software in textual form; thus, the task simply involved creating two text files, and initialising the adaptation layer with the correct file names.

The programming of the three application components was the largest task by a significant margin. The relative numbers of classes and lines of source code for the three components are shown in Table 7.1. It is interesting to note that, despite its relative complexity, the Negotiation component is substantially the smallest (and likewise required the least development effort). In implementing the Negotiation component, the programming toolkit was found to be reasonably easy to use, and the branching methods greatly simplified what would otherwise have been complex decision logic. The use of the branching programming model and its explicit preferences also afforded a degree of flexibility that would not have been possible by hardcoding the channel selection mechanism.

7.12 Discussion

The case study presented in this chapter served both to validate the utility of the modelling techniques, abstractions and software infrastructure developed in this thesis, and to highlight important topics for future research. A variety of results, derived from the requirements analysis, design and implementation phases described in this chapter, as well as subsequent informal experimentation involving the completed

⁵Note that, with appropriate software, the generation of the context database from the graphical model could be completely automated, if desired.

prototype, are summarised in this section.

Context modelling

Both context modelling solutions proved well suited to the task of capturing the types of context required by the channel selection problem. The fact-based modelling approach (CML) allowed the context requirements to be detailed in a formal manner which supported a natural and painless translation to a relational database. The case study also helped to validate the design principles upon which CML was founded - most notably, the importance of profiled context (such as interpersonal relationships) as a supplement to sensor-derived information.

The situation abstraction offered a very natural solution for describing contexts in general terms, and the ability to easily and arbitrarily combine situations to form rich context descriptions became particularly useful when defining preference scopes.

Preference modelling

The case study was especially important in evaluating the utility of the preference model, as it afforded an opportunity to experiment with a range of preference sets and empirically evaluate the correspondence between the desired and actual effects of preferences. The results were largely positive, with problems only arising in relation to preferences that assigned frequent indifferent scores. This occurred when the preference scopes incompletely covered the set of possible contexts (for example, when the user specified preferences only in relation to urgent interactions). Two simple solutions to this problem are possible:

1. to augment the preferences to provide complete coverage (as, for example, in the default recipient preference set in Figure 7.9, where the final preference is present only to ensure full coverage); or
2. to program the application to appropriately handle choices that receive indifferent scores (in the case of Mercury, this would likely mean permitting channels that receive indifferent scores as valid candidate channels).

The first alternative is preferable, particularly as the correct interpretation of indifferent scores in the second alternative is not always clear. Regardless of the option that is adopted, however, fine-tuning of the preferences upon completion of the implementation is always beneficial, in order to verify the absence of anomalous behaviour. This is especially important when the preference sets are large and complex.

Programming model and toolkit

The case study highlighted the particular importance of accurately capturing user requirements when developing context-aware software, and the value of the programming model's use of an explicit preference model in this regard. Indeed, one of the greatest benefits of the programming model seen in the case study was a decreased obligation on the part of the developer to perfectly understand and describe user requirements at the design phase, thanks to the ability to easily experiment with (and fine-tune) different preferences and context-aware behaviours following the implementation phase.

The success of the branching programming model was further demonstrated by the ease with which the negotiation protocol was implemented, and the considerable degree of flexibility that resulted. The desired loose coupling between the application logic, the context model and the user preferences was fully realised: of the three application components, only the negotiation component exhibits any dependence upon the context and preferences, and this is of a such a limited nature that even fairly radical changes to the context model and preferences are possible without corresponding modification of the implementation. Specifically, the negotiation component only directly queries the *HasChannel* fact type (in order to discover candidate channels); consequently, any other fact type can be removed or redefined with implications only on the definitions of the affected situation predicates (and possibly also on some of the preferences that are dependent upon these). New fact types can be added to the context model without requiring changes even to the situation predicates or preferences. The ability to change the context model in this way is extremely useful, as it enables the model to evolve as the supporting context sensing infrastructure changes, and allows the breadth of the user preferences to be similarly flexible.

As described in Section 7.11, the query and branching facilities of the programming toolkit were discovered to be powerful and easy to use, and the source code implementing the selection and sorting of candidate channels was consequently minimal and straightforward. However, several avenues remain open for future work. First, as mentioned in Section 6.3.4, there is considerable potential to optimise the evaluation of situation predicates and preferences. Although the current performance is completely adequate for Mercury's requirements, this may no longer be the case when the services of the adaptation layer must be shared between a large number of concurrently executing applications. Second, the branching toolkit has a current bias toward the retrieval and selection of choices that receive numerical and obligation scores. Further support for handling choices with indifferent and prohibition scores, and for performing error handling in response to undefined scores, could usefully be added.

7.13 Conclusions and future work

The results of the case study presented in this chapter are highly encouraging, and would be further strengthened by additional research in three areas. First, the evaluation of the developed prototype in realistic settings would enable a more accurate assessment of the user-friendliness of the preference model, and additionally help to identify other unanticipated usability issues. Second, further case studies, involving a variety of application domains, would allow the modelling approaches to be evaluated against a range of application and user requirements. A variety of candidate domains were highlighted in Section 1.5. Finally, the incorporation of live context information from sensors into the prototype would help to identify any issues related to imperfect, real-time data that may not yet have been adequately addressed. These research topics all fall beyond the scope of this thesis.

Chapter 8

Discussion and conclusions

This chapter concludes the thesis with a summary of research contributions and a brief discussion of key topics for future work.

8.1 Summary of contributions

Broadly, the aim of this thesis was to address the difficulties associated with the construction of context-aware computing applications, by developing a framework consisting of both sound conceptual foundations and a supporting software infrastructure. The context-aware design approach is currently within its infancy, with the present generation of applications largely being developed and trialled only in constrained laboratory settings. This situation persists as a result of high application development overheads, social barriers associated with privacy and usability, and an imperfect understanding of the truly compelling (and commercially viable) uses of context-awareness. The framework developed in this thesis addresses all of these challenges: the first by simplifying design and implementation tasks associated with context-aware software, and the latter two by facilitating the types of rapid prototyping and experimentation that are required in order to explore and overcome these obstacles.

The framework differs significantly from the previously proposed solutions described in Chapter 3, such as the Context Toolkit [95], as it is founded on a set of well-integrated abstractions that support the developer seamlessly through analysis, design, implementation and beyond. Specifically, the context model used for analysis and design tasks maps cleanly to a context management infrastructure that is exploited by the implementation. A similar lifecycle is supported by the situation, trigger and preference models. At design time, these allow generic context-aware behaviours to be identified and documented. These are realised during the implementation phase using the programming toolkit, and later fine-tuned by the developer and customised by the user at run-time.

The framework makes a number of important research contributions. These are summarised, by chapter, in the remainder of this section.

Chapter 2 motivated the need for a clear distinction between context information and the more nebulous concept of context, and argued that a firm understanding of the characteristics of the former are key to the design of successful context-aware systems. With this in mind, the chapter presented a novel characterisation of context information in terms of its sources, temporal and quality characteristics, and dependencies. The requirements arising from this characterisation form the foundations of the proposed framework, directly informing the design of the context modelling approach described in Chapter 4.

Chapter 3 presented a survey of related work, and identified an overwhelming lack of high-level design and programming tools suitable for the development of context-aware systems. In response, Chapters 4, 5 and 6 proposed a multi-tiered solution, integrating novel context modelling techniques, programming abstractions and software infrastructure.

Chapter 4 introduced a series of conceptual modelling constructs, in the form of the Context Modelling Language (CML), that can be used by the application designer to formulate a detailed graphical model of context requirements. The constructs are sufficiently abstract to meet the needs of this design task, while simultaneously being suitably formal to support run-time context management tasks. A straightforward mapping of the constructs to the relational data model was presented towards the end of the chapter, enabling the use of a relational DBMS as the basis for a context management infrastructure. The alignment of CML with modelling solutions long used within the field of information systems represents an important innovation, as it creates new opportunities for exploiting mature research solutions developed within this field, in relation to the modelling and management of temporal and real-time data, information quality, and uncertainty.

Chapter 5 identified a need for high-level programming abstractions built upon the context modelling approach of Chapter 4. It presented a solution for describing contexts in general terms, using the situation abstraction, as well as a pair of complementary programming models that exploit this abstraction to support both context-driven and application-driven context-aware behaviour.

The situation model provides a solution for describing context requirements in a selective, high-level manner, close to the way humans conceptualise context. Situations are expressed as conditions (captured by predicates) over a narrow set of parameters that are relevant to the task at hand. The use of the situation abstraction as a programming tool leads to a clean separation of application functionality from the fine-grained detail of the CML-based context model. This allows the latter to be easily evolved over time in response to changes in the sensing infrastructure and user requirements.

The context-driven programming model described in Chapter 5, based on triggering, is not novel, having been exploited previously in a variety of adaptive and context-aware applications. The contribution of this thesis in relation to the model was to demonstrate that a powerful new form of triggering that accommodates uncertain context information (Situation-Based Triggering) can be realised using the situation abstraction, by exploiting changes in situation states as the events upon which trigger actions are invoked.

In contrast, the branching programming model (Situation-Based Branching) provides a new approach for inserting context-dependent decision points into application logic, such that different paths or branches are executed depending upon the context. This type of behaviour is conventionally realised using primitive programming constructs, such as case or if statements. The branching model represents a considerably more flexible solution, however, as it implements choices as a function of both user preferences and context information. This approach affords the degree of user customisation and control required in order to achieve user acceptance of highly autonomous context-aware software, such as the communication application developed in Chapter 7.

The final contribution of Chapter 5 was the novel preference modelling solution used to realise the branching model. The proposed solution is both compatible with automated techniques for preference elicitation and evolution, and user-friendly, allowing individual user requirements to be expressed as fine-grained preferences that are easily combined incrementally to achieve complex behaviours.

Chapter 6 served to integrate the theoretical contributions of Chapters 4 and 5 into a system architecture incorporating the acquisition, management and querying of context information, the management and evaluation of situation expressions, triggers and user preferences, and toolkit support for the two programming models. The architecture addresses key challenges of pervasive systems (introduced in Chapter 1) in relation to autonomy, resource limitations, scalability, and dynamic aspects of computing environments and user requirements. Chapter 6 additionally presented a lightweight implementation of the architecture in the form of a programming toolkit and a context database. This prototype is valuable both as a form of validation for the design of the architecture, and as a testbed for the underlying theoretical foundations.

Finally, Chapter 7 presented important practical results gained through the use of the framework developed in Chapters 4 to 6 in the implementation of a context-aware communication application. It illustrated, by example, the process and issues involved in the development process, highlighted the success of the framework in achieving the desired degrees of flexibility and autonomy, and demonstrated the utility of being able to experiment with and fine-tune application behaviour at runtime, by modifying preferences.

8.2 Future work

The concluding sections of Chapters 4 to 7 earlier identified a variety of possible extensions to the original research presented in this thesis. These will not be repeated here. Instead, the remainder of this section outlines a set of key research directions for the field of context-awareness as a whole.

8.2.1 Context management

An essential topic for future research is that of context management. Usability problems arising in context-aware software as a result of privacy concerns and imperfect context information are well recognised. These can be alleviated in part by the use of appropriate management infrastructure that implements privacy policies and performs tasks such as conflict detection and resolution, and enforcement of integrity constraints. Such mechanisms are already in place in some location management systems, but, as demonstrated in Section 3.2, have not yet been adequately generalised to deal with broader and richer types of context information. This represents a difficult task, owing to the extremely heterogeneous types of context information present in many context-aware systems.

The architecture presented in Chapter 6 incorporated a variety of management functionality into the context reception and management layers, including conflict detection and implementation of privacy policies. However, with the exception of the enforcement of some of the integrity constraints introduced in Chapter 4, this functionality has not been implemented. Other management functions, such the use of context dependencies to detect and refresh stale context information, have also been described intermittently throughout this thesis, but not yet fully developed.

8.2.2 Design tools and methodologies

The survey of related work in Chapter 3 demonstrated that the design process associated with context-aware software is poorly understood, and that no mature design tools are available. The work presented in this thesis partially addressed this issue by providing a conceptual modelling approach that can be used as a basis for the exploration and specification of an application's context requirements, and preference and trigger modelling techniques that allow default context-aware behaviour to be expressed in high-level terms. The utility of these solutions in the design process was illustrated in the case study presented in Chapter 7.

However, these solutions by themselves do not address all of the novel design challenges faced by developers of context-aware software. As discussed in Section 3.4.2, there is also a requirement for interaction design techniques and tools that facilitate the design of user interfaces that address the various HCI challenges of per-

vasive computing environments, and overcome known usability problems introduced by context-awareness, in relation to transparency, consistency, predictability [204], privacy [175], and user control [117]. Moreover, improved techniques for eliciting user requirements, and mapping these to preference representations such as that proposed in Chapter 5, are needed, as are privacy modelling solutions that allow privacy requirements to be identified, modelled and melded into application design.

In addition, better understanding of the overall software engineering process associated with context-aware applications is required. This process and its attendant issues were considered informally in the case study presented in Chapter 7, but these remain to be explored in a more general sense, and then formalised as a well-defined software engineering methodology.

8.2.3 Evaluation

Finally, and perhaps most importantly, further evaluation of context-aware software in realistic settings is required. This is currently difficult, as seamless, large-scale pervasive computing has not arrived, and easy to use “plug-and-play” sensing components are not yet available for gathering rich types of context information. Additionally, a lack of development tools and methodologies, and of comprehensive infrastructural support for the management and use of context information, implies that there are high overheads associated with the development of sophisticated context-aware applications. However, this problem is partially overcome by the research presented in this thesis.

Empirical evaluation of context-aware software is important for several reasons. First, it is crucial in identifying new usability problems, and in exploring appropriate solutions to these. Second, it provides important validation and feedback in terms of the suitability of proposed design, context modelling and programming techniques, such as those developed in this thesis. Finally, it will eventually lead researchers to better understand the most compelling uses of context-awareness, enabling the development of applications that are capable of achieving widespread acceptance and use.

Appendix A

Syntax and semantics of the situation abstraction

A.1 Syntax

The set of well-formed situation formulas for a given context model with relations $\mathbf{R} = C \cup DC \cup FD$ can be defined recursively as follows. A term is a constant, c , in \mathbf{dom} , a variable, v , in \mathbf{var} , or a function mapping one or more terms to a constant in \mathbf{dom} . Functions consist of

- arithmetic expressions of the form $t_1 + t_2$, $t_1 - t_2$, $t_1 * t_2$ or $t_1 \div t_2$, where t_1 and t_2 are themselves terms;
- expressions of the form $f(t_1, \dots, t_n)$, where each of t_1, \dots, t_n is a term or an ordered tuple, $\langle t_1, \dots, t_i \rangle$, of terms, and $f \in \mathbf{F}$, where \mathbf{F} is the set of function names (disjoint from \mathbf{R}). Each function, f , is either user-defined or one of the special predefined functions. The predefined functions include:
 - the *id* function, which, given a relation name, p , and an ordered tuple of i terms (such that $i = \mathit{arity}(p)$ if $p \in C \cup DC$ or 2 if $p \in FD$) yields the identifier of the corresponding fact in the context instance (or a special null identifier, if the fact does not appear in the instance); and
 - the *timenow* function which returns the current date and time; and
 - the *isnull* function, which indicates whether the value of a given variable is null (unknown).

The base situation formulas over the set of relations, \mathbf{R} , include:

- atoms over \mathbf{R} , which are expressions of the form $p[t_1, \dots, t_n]$, where $p \in \mathbf{R}$, $n = \mathit{arity}(p)$ if $p \in C \cup DC$ (else $n = 2$ if $p \in FD$) and t_1, \dots, t_n are either terms or anonymous variables, represented by the underscore ($_$);

- expressions of the form $t_1 = t_2$, $t_1 \neq t_2$, $t_1 < t_2$, $t_1 \leq t_2$, $t_1 > t_2$ and $t_1 \geq t_2$, where t_1 and t_2 are terms.

The well-formed situation formulas over \mathbf{R} include the base formulas, as well as formulas of the form:

- $(\varphi \wedge \psi)$, where φ and ψ are well-formed formulas over \mathbf{R} ;
- $(\varphi \vee \psi)$, where φ and ψ are well-formed formulas over \mathbf{R} ;
- $\neg\varphi$, where φ is a well-formed formula over \mathbf{R} ;
- $\exists x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \varphi$, where $p \in \mathbf{R}$, t_1, \dots, t_n are terms or anonymous variables, $\{x_1, \dots, x_i\} \subseteq \{t_1, \dots, t_n\}$ and φ is a well-formed formula over \mathbf{R} ; and
- $\forall x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \varphi$, where $p \in \mathbf{R}$, t_1, \dots, t_n are terms or anonymous variables, $\{x_1, \dots, x_i\} \subseteq \{t_1, \dots, t_n\}$ and φ is a well-formed formula over \mathbf{R} .

The notion of free and bound variables is required to define situation predicates. An occurrence of a variable x in a formula φ is considered free iff

- φ is an atom; or
- $\varphi = (\psi \wedge \xi)$ and the occurrence of x is free in ψ or ξ ; or
- $\varphi = (\psi \vee \xi)$ and the occurrence of x is free in ψ or ξ ; or
- $\varphi = \exists y_1, \dots, y_i \bullet p[t_1, \dots, t_n] \bullet \psi$, x is distinct from each of y_1, \dots, y_i and the occurrence of x is free in ψ ; or
- $\varphi = \forall y_1, \dots, y_i \bullet p[t_1, \dots, t_n] \bullet \psi$, x is distinct from each of y_1, \dots, y_i , and the occurrence of x is free in ψ .

An occurrence of x in φ is bound if it is not free. The set of variables that have at least one free occurrence in φ is denoted $free(\varphi)$.

A situation predicate named S over a context model with relations \mathbf{R} is defined as follows:

$$S(v_1, \dots, v_n) : \varphi$$

where φ is a well-formed formula over \mathbf{R} and $\{v_1, \dots, v_n\}$ is exactly $free(\varphi)$.

A.2 Semantics

The concept of a *valuation* is useful when defining the semantics of situation formulas. For a finite set of variables, $V \subseteq \mathbf{var}$, a valuation v over V is a total function

from V to the set **dom** of constants. The notation $v|_U$ denotes the restriction of the domain of v to a set of variables, U , where $U \subseteq \text{dom}(v)$ ¹.

Given a valuation, v and a term t , the evaluation of t with respect to v is defined as follows.

$$\text{eval}(t, v) = \begin{cases} v(t) & \text{if } t \in \text{dom}(v), \\ t & \text{if } t \in \mathbf{dom} \text{ or } t \in (\mathbf{var} - \text{dom}(v)), \\ \text{eval}(t_1, v) + \text{eval}(t_2, v) & \text{if } t = (t_1 + t_2), \\ \text{eval}(t_1, v) - \text{eval}(t_2, v) & \text{if } t = (t_1 - t_2), \\ \text{eval}(t_1, v) * \text{eval}(t_2, v) & \text{if } t = (t_1 * t_2), \\ \text{eval}(t_1, v) \div \text{eval}(t_2, v) & \text{if } t = (t_1 \div t_2), \\ f(\text{eval}(t_1), \dots, \text{eval}(t_n)) & \text{if } t = f(t_1, \dots, t_n) \end{cases}$$

The semantics of a situation formula, φ , are determined as follows, according to the interpretation rules introduced in Section 4.6.9. Given a valuation v over $\text{free}(\varphi)$, a context instance I_I satisfies φ for v , denoted $I_I \models \varphi[v]$, iff

- $\varphi = p[t_1, \dots, t_n]$ is an atom, $p \notin A$ (that is, p is not an alternative relation), and for each t_i , $1 \leq i \leq n$, there exists a mapping, u_i , of the anonymous variable $(-)$ to a constant c_i in **dom** such that:
 - $\langle \text{eval}(t_1, v \cup \{u_1\}), \dots, \text{eval}(t_n, v \cup \{u_n\}) \rangle >$
 - is in $I_I(p)$; or
- $\varphi = p[t_1, \dots, t_n]$ is an atom, $p \in A$, and
 - for each t_i , $1 \leq i \leq n$, there exists a mapping, u_i , of the anonymous variable $(-)$ to a constant c_i in **dom** such that there is a tuple $f_1 = \langle \text{eval}(t_1, v \cup \{u_1\}), \dots, \text{eval}(t_n, v \cup \{u_n\}) \rangle >$ in $I_I(p)$; and
 - there is no distinct tuple, $f_2 \in I_I(p)$ that satisfies $f_1[pk - \text{altRole}(p)] = f_2[pk - \text{altRole}(p)]$, where pk is the primary key of p^2 ; or
- $\varphi = (t_1 = t_2)$ and $\text{eval}(t_1, v) = \text{eval}(t_2, v)$; or
- $\varphi = (t_1 \neq t_2)$ and $\text{eval}(t_1, v) \neq \text{eval}(t_2, v)$; or
- $\varphi = (t_1 < t_2)$ and $\text{eval}(t_1, v) < \text{eval}(t_2, v)$; or
- $\varphi = (t_1 \leq t_2)$ and $\text{eval}(t_1, v) \leq \text{eval}(t_2, v)$; or

¹Here, $\text{dom}(v)$ denotes the domain of the function v .

²These conditions can be summed up as follows: an atom over an alternative fact type is satisfied by a context instance exactly when it matches a fact in the corresponding relation, and that fact has no alternatives.

- $\varphi = (t_1 > t_2)$ and $eval(t_1, v) > eval(t_2, v)$; or
- $\varphi = (t_1 \geq t_2)$ and $eval(t_1, v) \geq eval(t_2, v)$; or
- $\varphi = (\psi \wedge \xi)$ and $I_I \models \psi[v|_{free(\psi)}]$ and $I_I \models \xi[v|_{free(\xi)}]$; or
- $\varphi = (\psi \vee \xi)$ and either $I_I \models \psi[v|_{free(\psi)}]$ or $I_I \models \xi[v|_{free(\xi)}]$; or
- $\varphi = \neg\psi$ and $I_I \not\models \psi[v]$ (according to the definition of $\not\models$ given below); or
- $\varphi = (\exists x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \psi)$ and for each x_j ($1 \leq j \leq i$), there exists a constant c_j in **dom**, such that $I_I \models (p[t_1, \dots, t_n])[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$ and $I_I \models \psi[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$; or
- $\varphi = (\forall x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \psi)$ and for every set of constants c_1, \dots, c_i (each in **dom**) for which $I_I \models (p[t_1, \dots, t_n])[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$ holds, $I_I \models \psi[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$ also holds.

If I_I does not satisfy φ for v , it is said to *partially satisfy* φ for v , denoted $I_I \vdash \varphi[v]$, provided the following conditions are met:

- $\varphi = p[t_1, \dots, t_n]$ is an atom and for each t_i , $1 \leq i \leq n$, there exists a mapping, u_i , of the anonymous variable ($_$) to a constant c_i in **dom** such that there is a tuple $\langle a_1, \dots, a_n \rangle$ in $I_I(p)$ with a_j equal to $eval(t_j, v \cup \{u_j\})$ or the special null value for $1 \leq j \leq n$; or
- $\varphi = (t_1 = t_2)$ and $eval(t_1, v) = eval(t_2, v)$; or
- $\varphi = (t_1 \neq t_2)$ and $eval(t_1, v) \neq eval(t_2, v)$; or
- $\varphi = (t_1 < t_2)$ and $eval(t_1, v) < eval(t_2, v)$; or
- $\varphi = (t_1 \leq t_2)$ and $eval(t_1, v) \leq eval(t_2, v)$; or
- $\varphi = (t_1 > t_2)$ and $eval(t_1, v) > eval(t_2, v)$; or
- $\varphi = (t_1 \geq t_2)$ and $eval(t_1, v) \geq eval(t_2, v)$; or
- $\varphi = (\psi \wedge \xi)$ and
 - $I_I \models \psi[v|_{free(\psi)}]$ or $I_I \vdash \psi[v|_{free(\psi)}]$; and
 - $I_I \models \xi[v|_{free(\xi)}]$ or $I_I \vdash \xi[v|_{free(\xi)}]$; or
- $\varphi = (\psi \vee \xi)$ and either $I_I \models \psi[v|_{free(\psi)}]$ or $I_I \vdash \psi[v|_{free(\psi)}]$ or $I_I \models \xi[v|_{free(\xi)}]$ or $I_I \vdash \xi[v|_{free(\xi)}]$; or
- $\varphi = \neg\psi$ and $I_I \vdash \psi[v]$ or $I_I \not\models \psi[v]$ (as defined below); or
- $\varphi = (\exists x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \psi)$ and for some constants c_1, \dots, c_i (each in **dom**)

- $I_I \models (p[t_1, \dots, t_n])[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$; and
- $I_I \models \psi[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$ or $I_I \vdash \psi[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$; or
- $\varphi = (\forall x_1, \dots, x_i \bullet p[t_1, \dots, t_n] \bullet \psi)$ and for each set of constants c_1, \dots, c_i (each in **dom**) for which $I_I \models (p[t_1, \dots, t_n])[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$ holds, $I_I \models \psi[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$ or $I_I \vdash \psi[v \cup \{(x_1, c_1), \dots, (x_i, c_i)\}]$ also holds.

When neither $I_I \models \varphi[v]$ nor $I_I \vdash \varphi[v]$ hold, I_I is unsatisfied by φ and v , written $I_I \not\models \varphi[v]$.

For a situation predicate $S(v_1, \dots, v_n) : \varphi$, the evaluation of S with respect to a context instance I_I and a valuation v on $\{v_1, \dots, v_n\}$ is defined as follows:

$$val(S, I_I, v) = \begin{cases} true & \text{if } I_I \models \varphi[v] \\ possibly\ true & \text{if } I_I \vdash \varphi[v] \\ false & \text{if } I_I \not\models \varphi[v] \end{cases}$$

A.3 Composite situations

Situation predicates can be formed from simpler predicates using logical negation, conjunction and disjunction. Predicates constructed in this manner are referred to as *composite* situation predicates. A composite situation predicate, CS , is defined using the following notation:

$$CS(v_1, \dots, v_n) : P$$

Here P is a valid predicate formula. The valid predicate formulas can be defined recursively as follows:

- $S(t_1, \dots, t_m)$, where S is a situation predicate (composite or ordinary) defined by $S(u_1, \dots, u_m) : P$ and each t_i ($1 \leq i \leq m$) is either a constant in **dom** or a variable in v_1, \dots, v_n ;
- $\neg P$, where P is a valid predicate formula;
- $P \wedge Q$, where P and Q are valid predicate formulas; or
- $P \vee Q$, where P and Q are valid predicate formulas.

The semantics of composite situations are straightforward. Intuitively, the result of combining a set of predicates is the same as that of combining their corresponding formulas to create an ordinary situation predicate. That is, the composite predicate $CS(v_1, \dots, v_n) : P$ is semantically equivalent to the non-composite predicate, $S(v_1, \dots, v_n) : form(P)$, where $form(P)$ is a situation formula defined as follows.

Assuming $\varphi/\{(x_1, y_1), \dots, (x_i, y_i)\}$ represents the formula formed by replacing each occurrence of variable x_j in φ with the variable or constant y_j ($1 \leq j \leq i$):

- if $P = S(t_1, \dots, t_m)$, where S is a non-composite predicate defined as $S(u_1, \dots, u_m) : \varphi$, $form(P) = \varphi/\{(u_1, t_1), \dots, (u_m, t_m)\}$;
- if $P = CS(t_1, \dots, t_m)$, where CS is a composite predicate defined as $CS(u_1, \dots, u_m) : Q$, $form(P) = form(Q)/\{(u_1, t_1), \dots, (u_m, t_m)\}$;
- if $P = \neg Q$, $form(P) = \neg form(Q)$;
- if $P = Q \wedge R$, $form(P) = form(Q) \wedge form(R)$; and
- if $P = Q \vee R$, $form(P) = form(Q) \vee form(R)$.

Thus, the evaluation of composite situations can be defined as follows. If

$$CS(v_1, \dots, v_n) : P$$

is a composite situation predicate, I_I is a context instance and v is a valuation on v_1, \dots, v_n :

$$val(CS, I_I, v) = \begin{cases} true & \text{if } I_I \models form(P)[v] \\ possibly\ true & \text{if } I_I \vdash form(P)[v] \\ false & \text{if } I_I \not\models form(P)[v] \end{cases}$$

A.4 The extended situation abstraction

This section amends the syntax and semantics of the situation abstraction, as defined in the previous sections, to incorporate the two special functions, *possibly* and *possiblynot*, which were introduced in Section 5.3.1.

In addition to the well-formed formulas outlined in Section A.1, the *extended situation abstraction* permits the following formulas over \mathbf{R} :

- *possibly*(φ), where φ is a well-formed formula over \mathbf{R} ; and
- *possiblynot*(φ), where φ is a well-formed formula over \mathbf{R} .

The notion of free variables is extended to cover these formulas as follows. An occurrence of a variable x in a formula φ is considered free if

- $\varphi = possibly(\psi)$ and the occurrence of x is free in ψ ; or
- $\varphi = possiblynot(\psi)$ and the occurrence of x is free in ψ .

The semantics of the new formulas of the extended situation abstraction are as follows. Given a valuation v over $free(\varphi)$, a context instance I_I satisfies φ for v ($I_I \models \varphi[v]$) iff:

- $\varphi = possibly(\psi)$ and either $I_I \models \psi[v]$ or $I_I \vdash \psi[v]$; or
- $\varphi = possiblynot(\psi)$ and either $I_I \not\models \psi[v]$ or $I_I \vdash \psi[v]$; or
- $I_I \models \varphi[v]$ according to the rules of Section A.2.

Note that the semantics of $I_I \vdash \varphi[v]$ and $I_I \not\models$ remain as before.

The *extended predicate formulas* are defined identically to the standard predicate formulas introduced in Section A.3, with the exception that the following are also valid:

- $possibly(P)$, where P is a valid extended predicate formula; and
- $possiblynot(P)$, where P is a valid extended predicate formula.

The function *form* is extended to cover these new formulas as follows:

- if $P = possibly(Q)$, $form(P) = possibly(form(Q))$; and
- if $P = possiblynot(Q)$, $form(P) = possiblynot(form(Q))$.

Bibliography

- [1] D. A. Norman. *The Invisible Computer: Why good products can fail, the personal computer is so complex, and information appliances are the solution.* MIT Press, Cambridge, Mass., 1998.
- [2] A. Harter, A. Hopper, P. Steggles, A. Ward and P. Webster. The anatomy of a context-aware application. In *5th International Conference on Mobile Computing and Networking (MOBICOM)*, 59–68. Seattle, August 1999.
- [3] R. José and N. Davies. Scalable and flexible location-based services for ubiquitous information access. In *1st International Symposium on Handheld and Ubiquitous Computing*, Volume 1707 of *Lecture Notes in Computer Science*, 52–66. Springer, 1999.
- [4] F. Hohl, U. Kubach, A. Leonhardi, K. Rothermel and M. Schwehm. Next century challenges: Nexus - an open global infrastructure for spatial-aware applications. In *5th International Conference on Mobile Computing and Networking (MOBICOM)*, 249–255. Seattle, August 1999.
- [5] Y. Chen, X. Chen, X. Ding, F. Rao and D. Liu. BlueLocator: Enabling enterprise location-based services. In *3rd International Conference on Mobile Data Management (MDM)*, 167–168. Singapore, January 2002.
- [6] P. Castro, P. Chiu, T. Kremenek and R. Muntz. A probabilistic room location service for wireless networked environments. In *3rd International Conference on Ubiquitous Computing*, Volume 2201 of *Lecture Notes in Computer Science*, 18–34. Springer, 2001.
- [7] A. Schmidt, A. Takaluoma and J. Mäntyjärvi. Context-aware telephony over WAP. *Personal Technologies*, 4(4):225–229, September 2000.
- [8] C. Schmandt. Everywhere messaging. In *1st International Symposium on Handheld and Ubiquitous Computing*, Volume 1707 of *Lecture Notes in Computer Science*, 22–27. Springer, 1999.

- [9] Y. Hård af Segerstad and P. Ljungstrand. Instant messaging with WebWho. *International Journal of Human Computer Studies*, 56(1):147–171, January 2002.
- [10] A. Ranganathan, R. Campbell, A. Ravi and A. Mahajan. ConChat: A context-aware chat program. *IEEE Pervasive Computing, Special Issue on Context-Aware Computing*, 1(3):51–57, July-September 2002.
- [11] K. Nagel, C. D. Kidd, T. O’Connell, A. K. Dey and G. D. Abowd. The family intercom: Developing a context-aware audio communication system. In *3rd International Conference on Ubiquitous Computing*, Volume 2201 of *Lecture Notes in Computer Science*, 176–183. Springer, 2001.
- [12] K. Cheverst, N. Davies, K. Mitchell and A. Friday. Experiences of developing and deploying a context-aware tourist guide: the GUIDE project. In *6th International Conference on Mobile Computing and Networking (MOBICOM)*, 20–31. Boston, August 2000.
- [13] G. D. Abowd, C. G. Atkeson, J. Hong, S. Long, R. Kooper and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. *Wireless Networks*, 3(5):421–433, 1997.
- [14] R. Oppermann and M. Specht. A context-sensitive nomadic exhibition guide. In *2nd International Symposium on Handheld and Ubiquitous Computing*, Volume 1927 of *Lecture Notes in Computer Science*, 127–142. Springer, 2000.
- [15] A. K. Dey, M. Futakawa, D. Salber and G. D. Abowd. The conference assistant: Combining context-awareness with wearable computing. In *3rd International Symposium on Wearable Computers (ISWC)*, 21–28. San Francisco, October 1999.
- [16] A. K. Dey and G. D. Abowd. CybreMinder: A context-aware system for supporting reminders. In *2nd International Symposium on Handheld and Ubiquitous Computing*, Volume 1927 of *Lecture Notes in Computer Science*, 172–186. Springer, 2000.
- [17] N. Marmasse and C. Schmandt. Location-aware information delivery with ComMotion. In *2nd International Symposium on Handheld and Ubiquitous Computing*, Volume 1927 of *Lecture Notes in Computer Science*, 157–171. Springer, 2000.
- [18] J. N. Weatherall and A. Hopper. Predator: A distributed location service and example applications. In *2nd International Workshop on Cooperative Buildings*, Volume 1670 of *Lecture Notes in Computer Science*, 127–139. Springer, 1999.

- [19] J. I. Hong and J. A. Landay. A context/communication information agent. *Personal and Ubiquitous Computing: Special Issue on Situated Interaction and Context-Aware Computing*, 5(1):78–81, February 2001.
- [20] N. Ryan, J. Pascoe and D. Morse. Fieldnote: a handheld information system for the field. In *1st International Workshop on TeleGeoProcessing*, 156–163. Lyon, 1999.
- [21] D. Salber, A. K. Dey and G. D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 434–441. Pittsburgh, May 1999.
- [22] G. Chen and D. Kotz. Context aggregation and dissemination in ubiquitous computing systems. In *4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*. Callicoon, June 2002.
- [23] A. Schmidt, K. A. Aidoo, A. Takaluoma, U. Tuomela, K. Van Laerhoven and W. Van de Velde. Advanced interaction in context. In *1st International Symposium on Handheld and Ubiquitous Computing*, Volume 1707 of *Lecture Notes in Computer Science*, 89–101. Springer, 1999.
- [24] M. Ebling, G. D. H. Hunt and H. Lei. Issues for context services for pervasive computing. In *Middleware 2001 Workshop on Middleware for Mobile Computing*. Heidelberg, November 2001.
- [25] R. Hull, P. Neaves and J. Bedford-Roberts. Towards situated computing. In *1st International Symposium on Wearable Computers (ISWC)*, 146–153. Cambridge, October 1997.
- [26] B. N. Schilit. *A Context-Aware System Architecture for Mobile Distributed Computing*. Ph.D. thesis, Columbia University, May 1995.
- [27] T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra and M. Spasojevic. People, places, things: Web presence for the real world. *Mobile Networks and Applications (MONET)*, 7(5):365–376, October 2002.
- [28] P. J. Brown, J. D. Bovey and X. Chen. Context-aware applications: From the laboratory to the marketplace. *IEEE Personal Communications*, 4(5):58–64, October 1997.
- [29] P. J. Brown. Triggering information by context. *Personal Technologies*, 2(1):1–9, September 1998.

- [30] D. Petrelli, E. Not, M. Zancanaro, C. Strapparava and O. Stock. Modelling and adapting to context. *Personal and Ubiquitous Computing, Special Issue on Situated Interaction and Context-Aware Computing*, 5(1):20–24, February 2001.
- [31] S. S. Yau, F. Karim, Y. Wang, B. Wang and S. K. S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing, Special Issue on Context-Aware Computing*, 1(3):33–40, July–September 2002.
- [32] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn and K. R. Walker. Agile application-aware adaptation for mobility. In *16th ACM Symposium on Operating System Principles*, 276–287. Saint Malo, 1997.
- [33] B. Schilit, N. Adams and R. Want. Context-aware computing applications. In *Workshop on Mobile Computing Systems and Applications*, 85–90. Santa Cruz, December 1994.
- [34] A. C. Huang, B. C. Ling, S. Ponnekanti and A. Fox. Pervasive computing: What is it good for? In *ACM International Workshop on Data Engineering for Wireless and Mobile Access*, 84–91. Seattle, August 1999.
- [35] A. Schmidt. Implicit human computer interaction through context. *Personal Technologies*, 4(2&3):191–199, June 2000.
- [36] P. Brown, W. Bursleson, M. Lamming, O.-W. Rahlff, G. Romano, J. Scholtz and D. Snowdon. Context-awareness: Some compelling applications. In *CHI 2000 Workshop on the What, Who, Where, When, Why and How of Context-Awareness*. April 2000.
- [37] M. Lamming and D. Bohm. SPECs: Personal pervasive systems. *IEEE Computer*, 36(6):109–111, June 2003.
- [38] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Department of Computer Science, Dartmouth College, November 2000.
- [39] K. S. Nagel and G. D. Abowd. Challenges in developing a context-aware audio communication system. In *Human Computer Interaction Consortium (HCIC)*. Fraser, January 2002.
- [40] K. Henriksen, J. Indulska and A. Rakotonirainy. Modeling context information in pervasive computing systems. In *1st International Conference on Pervasive Computing (Pervasive)*, Volume 2414 of *Lecture Notes in Computer Science*, 167–180. Springer, 2002.

- [41] R. Want, A. Hopper, V. Falcao and J. Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, 1992.
- [42] P. Ljungstrand. Context awareness and mobile phones. *Personal and Ubiquitous Computing*, 5(1):58–61, February 2001.
- [43] D. Siewiorek, A. Smailagic, J. Furukawa, N. Moraveji, K. Reiger and J. Shaffer. SenSay: A context-aware mobile phone. Technical report, School of Computer Science, Carnegie Mellon University, 2003.
- [44] B. N. Schilit, D. M. Hilbert and J. Trevor. Context-aware communication. *IEEE Wireless Communications*, 9(5):46–54, October 2002.
- [45] J.-M. Wams and M. Van Steen. Pervasive messaging. In *1st IEEE Conference on Pervasive Computing and Communications (PerCom)*, 495–504. Fort Worth, March 2003.
- [46] A. J. H. Peddemors, M. M. Lankhorst and J. de Heer. Presence, location, and instant messaging in a context-aware application framework. In *4th International Conference on Mobile Data Management (MDM)*, Volume 2574 of *Lecture Notes in Computer Science*, 325–330. Springer, 2003.
- [47] S. Mitchell, M. D. Spiteri, J. Bates and G. Coulouris. Context-aware multimedia computing in the intelligent hospital. In *9th ACM SIGOPS European Workshop*. Kolding, September 2000.
- [48] P. J. Brown and G. J. F. Jones. Context-aware retrieval: exploring a new environment for information retrieval and information filtering. *Personal and Ubiquitous Computing*, 5(4):253–263, December 2001.
- [49] G. J. F. Jones and P. J. Brown. Challenges and opportunities for context-aware retrieval on mobile devices. In *SIGIR Workshop on Mobile Personal Information Retrieval*, 47–56. Tampere, 2002.
- [50] E. Not, D. Petrelli, M. Sarini, O. Stock, C. Strapparava and M. Zancanaro. Hyper-navigation in the physical space: adapting presentations to the user and to the situational context. *The New Review of Hypermedia and Multimedia*, 4:33–46, 1998.
- [51] G. Benelli, A. Bianchi, P. Marti, E. Not and D. Sennati. HIPS: Hyperinteraction within the physical space. In *IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, 1075–1078. Florence, June 1999.

- [52] B. B. Bederson. Audio augmented reality: A prototype automated tour guide. In *ACM Conference on Human Factors and Computing Systems (CHI)*, 210–211. Denver, May 1995.
- [53] R. Oppermann and M. Specht. Adaptive support for a mobile museum guide. In *Workshop on Interactive Applications of Mobile Computing (IMC)*. Rostock, November 1998.
- [54] M. Beigl. Memoclip: A location-based remembrance appliance. *Personal Technologies*, 4(4):230–233, September 2000.
- [55] S. Helal, B. Winkler, C. Lee, Y. Kaddourah, L. Ran, C. Giraldo and W. Mann. Enabling location-aware pervasive computing applications for the elderly. In *1st IEEE Conference on Pervasive Computing and Communications (PerCom)*. Fort Worth, March 2003.
- [56] S. S. Intille, K. Larson and C. Kukla. Just-in-time context-sensitive questioning for preventative health care. In *AAAI Workshop on Automation as Caregiver: The Role of Intelligent Technology in Elder Care*. July 2002.
- [57] E. Mynatt, I. Essa and W. Rogers. Increasing the opportunities for aging in place. In *ACM Conference on Universal Usability*, 65–71. Arlington, November 2000.
- [58] C. D. Kidd, R. J. Orr, G. D. Abowd, C. G. Atkeson, I. A. Essa, B. MacIntyre, E. Mynatt, T. E. Starner and W. Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *2nd International Workshop on Cooperative Buildings (CoBuild)*, Volume 1670 of *Lecture Notes in Computer Science*, 191–198. 1999.
- [59] V. Stanford. Using pervasive computing to deliver elder care. *IEEE Pervasive Computing*, 1(1):10–13, January-March 2002.
- [60] R. Beckwith and S. Lederer. Designing for one’s dotage: UbiComp and residential care facilities. In *Conference on Home Oriented Informatics and Telematics*. Irvine, April 2003.
- [61] R. Beckwith. Designing for ubiquity: The perception of privacy. *IEEE Pervasive Computing*, 2(2):40–46, April-June 2003.
- [62] M. C. Mozer. The neural network house: An environment that adapts to its inhabitants. In *AAAI Spring Symposium on Intelligent Environments*, 110–114. AAAI Press, 1998.

- [63] M. C. Mozer. An intelligent environment must be adaptive. *IEEE Intelligent Systems and their Applications*, 14(2):11–13, 1999.
- [64] V. Lesser, M. Atighetchi, B. Benyo, B. Horling, A. Raja, R. Vincent, T. Wagner, P. Xuan and S. X. Q. Zhang. The intelligent home testbed. In *Autonomy Control Software Workshop*. Seattle, January 1999.
- [65] Y. Shi, W. Xie, G. Xu, R. Shi, E. Chen, Y. Mao and F. Liu. The smart classroom: Merging technologies for seamless tele-education. *IEEE Pervasive Computing*, 2(2):47–55, April-June 2003.
- [66] G. D. Abowd. Software engineering issues for ubiquitous computing. In *21st International Conference on Software Engineering (ICSE)*, 78–84. Los Angeles, May 1999.
- [67] J. A. Brotherton. *Enriching Everyday Experiences through the Automated Capture and Access of Live Experiences - eClass: Building, Observing and Understanding the Impact of Capture and Access in an Educational Domain*. Ph.D. thesis, College of Computing, Georgia Institute of Technology, December 2001.
- [68] T. Hammond, K. Gajos, R. Davis and H. Shrobe. An agent-based system for capturing and indexing software design meetings. In *International Workshop on Agents in Design (WAID)*. Cambridge, August 2002.
- [69] A. K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, February 2001.
- [70] P. J. Brown. The stick-e document: a framework for creating context-aware applications. In *Electronic Publishing*, 259–272. Palo Alto, 1996.
- [71] S. Benford, R. Anastasi, M. Flintham, A. Drozd, A. Crabtree, C. Greenhalgh, N. Tandavanitj, M. Adams and J. Row-Farr. Coping with uncertainty in a location-based game. *IEEE Pervasive Computing*, 2(3):34–41, July-September 2003.
- [72] M. Flintham, R. Anastasi, S. Benford, T. Hemmings, A. Crabtree, C. Greenalgh, T. Rodden, N. Tandavanitj, M. Adams and J. Row-Farr. Where on-line meets on-the-streets: experiences with mobile mixed reality games. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 569–576. Florida, April 2003.
- [73] D. Narayanan, J. Flinn and M. Satyanarayanan. Using history to improve mobile application adaptation. In *3rd IEEE Workshop on Mobile Computing Systems and Applications*, 30–41. Monterey, August 2000.

- [74] D. Salber and G. D. Abowd. The design and use of a generic context server. In *Workshop on Perceptual User Interfaces (PUI)*, 63–66. San Francisco, November 1998.
- [75] C. Efstratiou, K. Cheverst, N. Davies and A. Friday. An architecture for the effective support of adaptive context-aware applications. In *2nd International Conference on Mobile Data Management (MDM)*, Volume 1987 of *Lecture Notes in Computer Science*, 15–26. Springer, 2001.
- [76] G. Widmer and M. Kubat. Effective learning in dynamic environments by explicit context tracking. In *6th European Conference on Machine Learning*, Volume 667 of *Lecture Notes in Artificial Intelligence*, 227–243. Springer, 1993.
- [77] P. D. Turney. The management of context-sensitive features: A review of strategies. In *13th International Conference on Machine Learning, Workshop on Learning in Context-Sensitive Domains*, 60–66. Bari, July 1996.
- [78] P. Brézillon and J.-C. Pomerol. User acceptance of interactive systems: Lessons from knowledge-based and decision support. *International Journal of Failure and Lessons Learned in Information Technology Management*, 1(1):67–75, 1997.
- [79] T. Toth-Fejel and D. Heher. Temporal and contextual knowledge in model-based expert systems. In *3rd Annual Conference on Artificial Intelligence for Space Applications*. Huntsville, 1987.
- [80] E. Walther, H. Eriksson and M. A. Musen. Plug-and-play: Construction of task-specific expert-system shells using sharable context ontologies. In *AAAI Workshop on Knowledge Representation Aspects of Knowledge Acquisition*, 191–198. San Jose, 1992.
- [81] R. P. Arritt and R. M. Turner. Situation assessment for autonomous underwater vehicles using a priori contextual knowledge. In *13th International Symposium on Unmanned Untethered Submersible Technology*. 2003.
- [82] D. Lenat. The dimensions of context-space. Technical report, Cycorp, 1998.
- [83] J. McCarthy. Notes on formalizing contexts. In *5th National Conference on Artificial Intelligence*, 555–560. Morgan Kaufmann, 1986.
- [84] S. Buvac and I. A. Mason. Propositional logic of context. In *11th National Conference on Artificial Intelligence*, 412–419. AAAI Press, 1993.
- [85] F. Giunchiglia and C. Ghidini. Local models semantics, or contextual reasoning = locality + compatibility. In *6th International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 282–291. Trento, June 1998.

- [86] J. H. Connolly. Context in the study of human languages and computer languages: a comparison. In *3rd International and Interdisciplinary Conference on Modeling and Using Context*, Volume 2116 of *Lecture Notes in Artificial Intelligence*, 116–128. Springer, 2001.
- [87] T. Bauer and D. B. Leake. WordSieve: a method for real-time context extraction. In *3rd International and Interdisciplinary Conference on Modeling and Using Context*, Volume 2116 of *Lecture Notes in Artificial Intelligence*, 30–44. Springer, 2001.
- [88] P. J. Brown and G. J. F. Jones. Exploiting contextual change in context-aware retrieval. In *17th ACM Symposium on Applied Computing (SAC)*, 650–656. Madrid, March 2002.
- [89] A. Ward, A. Jones and A. Hopper. A new location technique for the active office. *IEEE Personal Communications*, 4(5):42–47, October 1997.
- [90] U. Leonhard, J. Magee and P. Dias. Location service in mobile computing environments. *Computer and Graphics, Special Issue on Mobile Computing*, 20(5), September/October 1996.
- [91] P. Couderc and A.-M. Kermarrec. Enabling context-awareness from network-level location tracking. In *1st International Symposium on Handheld and Ubiquitous Computing*, Volume 1707 of *Lecture Notes in Computer Science*, 67–73. Springer, 1999.
- [92] N. B. Priyantha, A. Chakraborty and H. Balakrishnan. The Cricket location-support system. In *6th Annual ACM International Conference on Mobile Computing and Networking*, 32–43. Boston, August 2000.
- [93] P. Bahl and V. N. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *IEEE Conference on Computer Communications (INFOCOM)*, 775–784. Tel-Aviv, March 2000.
- [94] R. J. Orr and G. D. Abowd. The Smart Floor: A mechanism for natural user identification and tracking. In *ACM Conference on Human Factors in Computing Systems (CHI)*. The Hague, April 2000.
- [95] A. K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. Ph.D. thesis, College of Computing, Georgia Institute of Technology, December 2000.
- [96] P. Gray and D. Salber. Modelling and using sensed context information in the design of interactive applications. In *8th IFIP International Conference on*

- Engineering for Human-Computer Interaction*, Volume 2254 of *Lecture Notes in Computer Science*, 317–336. Springer, 2001.
- [97] A. Schmidt and K. Van Laerhoven. How to build smart appliances. *IEEE Personal Communications, Special Issue on Pervasive Computing*, 8(4):66–71, August 2001.
- [98] H.-W. Gellersen, A. Schmidt and M. Beigl. Multi-sensor context-awareness in mobile devices and smart artifacts. *Mobile Networks and Applications (MONET)*, 7(5):341–351, October 2002.
- [99] K. Van Laerhoven, K. A. Aidoo and S. Lowette. Real-time analysis of data from many sensors with neural networks. In *5th International Symposium on Wearable Computers (ISWC)*, 115–122. Zurich, October 2001.
- [100] G. Chen and D. Kotz. Solar: An open platform for context-aware mobile applications. In *1st International Conference on Pervasive Computing (Pervasive), Short Paper Proceedings*, 41–47. Zurich, August 2002.
- [101] G. Chen and D. Kotz. Context-sensitive resource discovery. In *1st IEEE Conference on Pervasive Computing and Communications (PerCom)*, 243–252. Fort Worth, March 2003.
- [102] B. N. Schilit, M. M. Theimer and B. B. Welch. Customizing mobile applications. In *USENIX Symposium on Mobile and Location-Independent Computing*, 129–138. Cambridge, August 1993.
- [103] J. Pascoe. Adding generic contextual capabilities to wearable computers. In *2nd International Symposium on Wearable Computers (ISWC)*, 92–99. Pittsburgh, October 1998.
- [104] G. Judd and P. Steenkiste. Providing contextual information to pervasive computing applications. In *1st IEEE Conference on Pervasive Computing and Communications (PerCom)*, 133–142. Fort Worth, March 2003.
- [105] H. Lei, D. M. Sow, J. S. Davis, G. Banavar and M. R. Ebling. The design and applications of a context service. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):45–55, October 2002.
- [106] J. Myllymaki and S. Edlund. Location aggregation from multiple sources. In *3rd International Conference on Mobile Data Management (MDM)*, 131–138. Singapore, January 2002.
- [107] J. Indulska, T. McFadden, M. Kind and K. Henriksen. Scalable location management for context-aware systems. In *4th IFIP International Conference*

- on Distributed Applications and Interoperable Systems (DAIS) (to appear)*. Paris, November 2003.
- [108] G. Myles, A. Friday and N. Davies. Preserving privacy in environments with location-based applications. *IEEE Pervasive Computing*, 2(1):56–64, January–March 2003.
- [109] J. Pascoe, N. S. Ryan and D. R. Morse. Human-Computer-Giraffe interaction: HCI in the field. In *Workshop on Human Computer Interaction with Mobile Devices*. Glasgow, May 1998.
- [110] G. J. F. Jones and P. J. Brown. Context-aware retrieval for pervasive computing environments. In *1st International Conference on Pervasive Computing - Short Paper Proceedings*, 10–27. Zurich, August 2002.
- [111] A. K. Dey, D. Salber and G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2-4):97–166, 2001.
- [112] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell and K. Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing, Special Issue on Wearable Computing*, 1(4):74–83, October–December 2002.
- [113] T. Y. Sohn and A. K. Dey. iCAP: An informal tool for interactive prototyping of context-aware applications. In *ACM Conference on Human Factors in Computing Systems (CHI)*, 974–975. ACM Press, Ft. Lauderdale, April 2003.
- [114] J. Coutaz and G. Rey. Recovering foundations for a theory of contextors. In *4th International Conference on Computer-Aided Design of User Interfaces (CADUI)*. Valenciennes, May 2002.
- [115] K. Cheverst, N. Davies, K. Mitchell and C. Efstratiou. Using context as a crystal ball: Rewards and pitfalls. *Personal and Ubiquitous Computing*, 5(1):8–11, February 2001.
- [116] C. Lueg. On the gap between vision and feasibility. In *1st International Conference on Pervasive Computing (Pervasive)*, Volume 2414 of *Lecture Notes in Computer Science*, 45–57. Springer, 2002.
- [117] T. Erickson. Some problems with the notion of context-aware computing. *Communications of the ACM*, 45(2):102–104, February 2002.
- [118] H. E. Byun and K. Cheverst. Exploiting user models and context-awareness to support personal daily activities. In *UM2001 Workshop on User Modeling for Context-Aware Applications*. Sonthofen, July 2001.

- [119] H. E. Byun and K. Cheverst. Harnessing context to support proactive behaviours. In *ECAI2002 Workshop on AI in Mobile Systems*. Lyon, July 2002.
- [120] A. Jameson. Modeling both the context and the user. *Personal and Ubiquitous Computing*, 5(1):29–33, February 2001.
- [121] L. Marucci and F. Paternò. Supporting adaptivity to heterogeneous platforms through user models. In *4th International Symposium on Mobile Human-Computer Interaction (Mobile HCI)*, Volume 2411 of *Lecture Notes in Computer Science*, 409–413. Springer, 2002.
- [122] M. Samulowitz. Towards context-aware user modelling. In *3rd International IFIP/GI Working Conference on Trends in Distributed Systems: Towards a Universal Service Market*, Volume 1890 of *Lecture Notes in Computer Science*, 272–277. Springer, 2000.
- [123] C. Schmandt, N. Marmasse, S. Marti, N. Sawhney and S. Wheeler. Everywhere messaging. *IBM Systems Journal*, 39(3&4):660–677, 2000.
- [124] M. Nilsson, J. Hjelm and H. Ohto. Composite capabilities/preference profiles: Requirements and architecture, 21 July 2000. W3C Working Draft.
- [125] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, J. Hjelm, M. H. Butler and L. Tran. Composite capability/preference profiles (CC/PP): Structure and vocabularies 1.0, 15 October 2003. W3C Proposed Recommendation.
- [126] O. Lassila and R. R. Swick. Resource description framework (RDF) model and syntax specification. W3C Recommendation, 22 February 1999.
- [127] P. Fishburn. Preference structures and their numerical representations. *Theoretical Computer Science*, 217(2):359–383, 1999.
- [128] R. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Wiley, 1976.
- [129] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *ACM SIGMOD Conference on Management of Data*, 297–306. Dallas, May 2000.
- [130] J. Chomicki. Querying with intrinsic preferences. In *8th International Conference on Extending Database Technology (EDBT)*, Volume 2287 of *Lecture Notes in Computer Science*, 34–51. Springer, 2002.
- [131] W. W. Cohen, R. E. Schapire and Y. Singer. Learning to order things. *Advances in Neural Information Processing Systems*, 10, 1998.

- [132] U. Çetintemel, M. J. Franklin and C. Lee Giles. Self-adaptive user profiles for large-scale data delivery. In *16th International Conference on Data Engineering (ICDE)*, 622–633. San Diego, February-March 2000.
- [133] L. Cranor, M. Langheinrich and M. Marchiori. A P3P preference exchange language 1.0 (APPEL1.0). W3C Working Draft, 15 April 2002.
- [134] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall and J. Reagle. The platform for privacy preferences 1.0 (P3P1.0) specification. W3C Recommendation, 16 April 2002.
- [135] J. I. Hong and J. A. Landay. An infrastructure approach to context-aware computing. *Human-Computer Interaction*, 16(2&3), 2001.
- [136] T. A. Halpin. *Conceptual Schema and Relational Database Design*, chapter 2. Prentice Hall Australia, Sydney, 2nd edition, 1995.
- [137] H. Gregersen and C. S. Jensen. Temporal entity-relationship models - a survey. *IEEE Transactions on Knowledge and Data Engineering*, 11(3):464–497, 1999.
- [138] R. Wang, M. P. Reddy and H. Kon. Towards quality data: An attribute-based approach. *Decision Support Systems*, 13:349–372, 1995.
- [139] V. Storey and R. Wang. Modeling quality requirements in conceptual database design. In *3rd Conference on Information Quality (IQ)*, 64–87. Cambridge, October 1998.
- [140] A. Datta. Databases for active rapidly changing data systems (ARCS): Augmenting real-time databases with temporal and active characteristics. In *1st International Workshop on Real-Time Databases: Issues and Applications*, 7–13. California, March 1996.
- [141] T. Imielinski and W. Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, October 1984.
- [142] T. Imielinski and W. Lipski. The relational model of data and cylindric algebras. *Journal of Computer and System Sciences*, 28(1):80–102, 1984.
- [143] M. Y. Vardi. On the integrity of databases with incomplete information. In *5th ACM Symposium on Principles of Database Systems*, 252–266. Cambridge, March 1986.
- [144] M. Y. Vardi. Querying logical databases. *Journal of Computer and System Sciences*, 33(2):142–160, October 1986.

- [145] J. M. Morrissey. Imprecise information and uncertainty in information systems. *ACM Transactions on Information Systems*, 8(2):159–180, April 1990.
- [146] Q. Kong and G. Chen. On deductive database with incomplete information. *ACM Transactions on Information Systems*, 13(3):355–369, July 1995.
- [147] Object Management Group (OMG). Unified Modeling Language specification v1.5. OMG document formal/03-03-01, 2003.
- [148] P. P.-S. S. Chen. The Entity-Relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
- [149] T. A. Halpin. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. Morgan Kaufman, San Francisco, 2001.
- [150] G. M. Nijssen and T. A. Halpin. *Conceptual Schema and Relational Database Design: A Fact Oriented Approach*. Prentice Hall, New York, 1989.
- [151] T. A. Halpin. *Conceptual Schema and Relational Database Design*. Prentice Hall Australia, Sydney, 2nd edition, 1995.
- [152] C. S. Jensen and R. T. Snodgrass. Semantics of time-varying attributes and their use for temporal database design. In *14th International Conference on Object-Oriented and Entity-Relationship Modelling*, Volume 1021 of *Lecture Notes in Computer Science*, 366–377. Springer, 1995.
- [153] J. Chomicki. Temporal query languages: A survey. In *1st International Conference on Temporal Logic (ICTL)*, Volume 827 of *Lecture Notes in Computer Science*, 506–534. Springer, 1994.
- [154] C. S. Jensen et al. The consensus glossary of temporal database concepts - February 1998 version. In *Temporal Databases: Research and Practice*, Volume 1399 of *Lecture Notes in Computer Science*, 367–405. Springer, 1998.
- [155] B. Tausovitch. Towards temporal extensions to the entity-relationship. In *10th International Conference on the Entity-Relationship Approach (ER)*, 163–179. San Mateo, October 1991.
- [156] J. Wijsen and R. T. Ng. Temporal dependencies generalized for spatial and other dimensions. In *International Workshop on Spatio-Temporal Database Management*, Volume 1678 of *Lecture Notes in Computer Science*, 189–203. Springer, 1999.
- [157] A. K. Dey, J. Mankoff, G. D. Abowd and S. A. Carter. Distributed mediation of ambiguous context in aware environments. In *15th Annual Symposium on User Interface Software and Technology (UIST)*, 121–130. Paris, October 2002.

- [158] N. Bulusu, D. Estrin and J. Heidemann. Tradeoffs in location support systems: The case for quality-expressive location models for applications. In *Workshop on Location Modeling, 3rd International Conference on Ubiquitous Computing*. Atlanta, September 2001.
- [159] J. Hightower and G. Borriello. Real-time error in location modeling for ubiquitous computing. In *Workshop on Location Modeling, 3rd International Conference on Ubiquitous Computing*. Atlanta, September 2001.
- [160] H. E. Byun and K. Cheverst. Supporting proactive ‘intelligent’ behaviour: the problem of uncertainty. In *1st International Workshop on User Modelling for Ubiquitous Computing*. Pittsburgh, June 2003.
- [161] F. Bacchus. *Representing and reasoning with probabilistic knowledge: a logical approach to probabilities*. MIT Press, Cambridge, Mass., 1991.
- [162] D. Dubois, J. Lang and H. Prade. *Possibilistic Logic*, Volume 3 of *Handbook of Logic in Artificial Intelligence and Logic Programming*, 439–513. Oxford University Press, 1994.
- [163] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 3rd edition, 1998.
- [164] V. Biazzo, R. Giugno, T. Lukasiewicz and V. S. Subrahmanian. Temporal probabilistic object bases. *IEEE Transactions on Knowledge and Data Engineering*, 15(4):921–939, July/August 2003.
- [165] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
- [166] S. Abiteboul, R. Hull and V. Vianu. *Foundations of databases*. Addison-Wesley, Reading, Mass., 1995.
- [167] R. Elmasri and S. B. Navathe. *Fundamentals of database systems*. Addison-Wesley, Reading, Mass., 3rd edition, 2000.
- [168] R. Reiter. *On closed world data bases*, 55–76. *Logic and Data Bases*. Plenum, New York, 1978.
- [169] X. Jiang, J. I. Hong and J. A. Landay. Approximate information flows: Socially-based modeling of privacy in ubiquitous computing. In *4th International Conference on Ubiquitous Computing*, Volume 2498 of *Lecture Notes in Computer Science*, 176–193. Springer, 2002.

- [170] M. Langheinrich. Privacy by design - principles of privacy-aware ubiquitous systems. In *3rd International Conference on Ubiquitous Computing*, Volume 2201 of *Lecture Notes in Computer Science*, 273–291. Springer, 2001.
- [171] M. Langheinrich. A privacy awareness system for ubiquitous computing environments. In *4th International Conference on Ubiquitous Computing*, Volume 2498 of *Lecture Notes in Computer Science*, 237–245. Springer, 2002.
- [172] M. J. Covington, W. Long, S. Srinivasan, A. K. Dey, M. Ahamad and G. D. Abowd. Securing context-aware applications using environment roles. In *6th ACM Symposium on Access Control Models and Technologies (SACMAT)*, 10–20. Chantilly, May 2001.
- [173] M. J. Covington, P. Fogla, Z. Zhan and M. Ahamad. A context-aware security architecture for emerging applications. In *Annual Computer Security Applications Conference (ACSAC)*. Las Vegas, December 2002.
- [174] A. Smailagic, D. P. Siewiorek, J. Anhalt, D. Kogan and Y. Wang. Location sensing and privacy in a context aware computing environment. *Pervasive Computing*, 2001.
- [175] X. Jiang and J. A. Landay. Modeling privacy control in context-aware systems. *IEEE Pervasive Computing*, 1(3):59–63, July-September 2002.
- [176] F. L. Gandon and N. M. Sadeh. A semantic e-Wallet to reconcile privacy and context awareness. In *2nd International Semantic Web Conference*. Florida, October 2003.
- [177] P. Robinson and M. Beigl. Trust context spaces: An infrastructure for pervasive security in context-aware environments. In *1st International Conference on Security in Pervasive Computing*, Lecture Notes in Computer Science, 157–172. Springer, 2004.
- [178] R. Agrawal, J. Kiernan, R. Srikant and Y. Xu. Hippocratic databases. In *28th International Conference on Very Large Data Bases (VLDB)*, 143–154. Hong Kong, 2002.
- [179] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman and D. Zukowski. Challenges: An application model for pervasive computing. In *6th International Conference on Mobile Computing and Networking (MOBICOM)*, 266–274. Boston, August 2000.
- [180] R. K. Balan, J. P. Sousa and M. Satyanarayanan. Meeting the software engineering challenges of adaptive mobile applications. Technical Report CMU-

- CS-03-111, School of Computer Science, Carnegie Mellon University, February 2003.
- [181] D. Garlan and B. Schmerl. Component-based software engineering in a pervasive computing environment. In *4th ICSE Workshop on Component-Based Software Engineering*. Toronto, May 2001.
- [182] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello and D. Wetherall. Programming for pervasive computing environments. Technical Report UW-CSE-01-06-01, Department of Computer Science and Engineering, University of Washington, June 2001.
- [183] K. Cheverst, K. Mitchell and N. Davies. Exploring context-aware information push. *Personal and Ubiquitous Computing*, 6(4):276–281, September 2002.
- [184] L. Barkhuus and A. K. Dey. Is context-aware computing taking control away from the user? Three levels of interactivity examined. In *5th International Conference on Ubiquitous Computing*. Seattle, October 2003.
- [185] E. Kaasinen. User needs for location-aware mobile services. *Personal and Ubiquitous Computing*, 7(1):70–79, May 2003.
- [186] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Francisco, 1993.
- [187] G. Restall. Laws of non-contradiction, laws of the excluded middle and logics. In *The Law of Non-Contradiction; New Philosophical Essays*. Oxford University Press, (to appear).
- [188] U. Dayal, B. T. Blaustein, A. P. Buchmann, U. S. Chakravarthy, M. Hsu, R. Ledin, D. R. McCarthy, A. Rosenthal, S. K. Sarin, M. J. Carey, M. Livny and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *ACM SIGMOD Record*, 17(1):51–70, March 1988.
- [189] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *17th International Conference on Very Large Data Bases (VLDB)*, 327–336. Barcelona, September 1991.
- [190] J.-J. Meyer and R. J. Wieringa. Deontic logic: A concise overview. In *Deontic Logic in Computer Science: Normative System Specification*, 3–16. John Wiley and Sons, England, 1993.
- [191] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

- [192] D. Petrelli, A. De Angeli and G. Convertino. A user-centered approach to user modeling. In *7th International Conference on User Modeling (UM)*, 255–264. Banff, June 1999.
- [193] G. Linden, S. Hanks and N. Lesh. Interactive assessment of user preference models: The automated travel assistant. In *6th International Conference on User Modeling (UM)*, 67–78. Chia Laguna, June 1997.
- [194] V. A. Ha and P. Haddawy. Toward case-based preference elicitation: Similarity measures on preference structures. In *14th Conference on Uncertainty in Artificial Intelligence*, 193–201. Madison, July 1998.
- [195] P. Sutton, R. Arkins and B. Segall. Supporting disconnectedness - transparent information delivery for mobile and invisible computing. In *1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, 277–285. Brisbane, May 2001.
- [196] Object Management Group (OMG). Common Object Request Broker Architecture (CORBA/IIOP) Specification, v3.0.2. OMG document formal/2002-12-02, 2002.
- [197] PostgreSQL: An open-source object relational database management system (ORDBMS). Technical report, Paragon Corporation, June 2002.
- [198] PostgreSQL online documentation.
URL: www.postgresql.org
- [199] JavaCC online documentation.
URL: javacc.dev.java.net
- [200] M. Ackerman, T. Darrell and D. J. Weitzner. Privacy in context. *Human Computer Interaction*, 16(2):167–176, 2001.
- [201] S. Parsowith, G. Fitzpatrick, S. Kaplan, B. Segall and J. Boot. Tickertape: Notification and communication in a single line. In *Asia Pacific Computer Human Interaction (APCHI)*, 139–144. Kangawa, July 1998.
- [202] N. Sawhney and C. Schmandt. Nomadic radio: speech and audio interaction for contextual messaging in nomadic environments. *ACM Transactions on Computer-Human Interaction*, 7(3):353–383, 2000.
- [203] JDK version 1.4.2 online documentation.
URL: <http://java.sun.com/j2se/1.4.2/>

- [204] K. Cheverst, N. Davies, K. Mitchell and A. Friday. The role of connectivity in supporting context-sensitive applications. In *1st International Symposium on Handheld and Ubiquitous Computing*, Volume 1707 of *Lecture Notes in Computer Science*, 193–207. Springer, 1999.