An Agent-Based Model for the Development of Intelligent Mobile Services

SIKS Dissertation Series No. 2009-26

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

ISBN: 978-90-39350805

An Agent-Based Model for the Development of Intelligent Mobile Services

Intelligente Mobiele Diensten met behulp van
Agent Technologie
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht
op gezag van de rector magnificus, prof. dr. J.C. Stoof,
ingevolge het besluit van het college voor promoties
in het openbaar te verdedigen
op maandag 5 oktober 2009 des middags te 12.45 uur

door

Fernando Luiz Koch

geboren op 19 augustus 1971, te Curitiba-PR, Brazilië

Promotoren:	Prof. dr. J.-J. Ch. Meyer
	Prof. dr. E. Sonenberg
Co-promotor:	Dr. F. Dignum

Contents

Preface

I had two main objectives in doing my Ph.D.: (i) to demonstrate to myself that I am capable of acquiring a higher degree, and (ii) to conduct research that would have me thinking of conceptual problems and on "how people think". The choice for mobile services was a pragmatical one: as I have been working in the mobile computing industry for several years, thus I thought of doing something to contribute to this area.

As it is usual to Ph.D. thesis prefaces, I echo others saying that writing a Ph.D thesis is difficult, sometimes frustrating, and unimaginably laborious. I experienced that in two ways. First, what I describe as an "accumulation of small frustrations". It took me into many areas of scientific and philosophical inquiry that, in practice, led me to take days to get "the right sentence"; a task I usually complete in hours. Moreover, I felt it was necessary staying in touch with reality. Hence, besides taking forever to write one sentence down, I was always asking: "is this real?"; "could this be part of a real solution?", or "would this be useful after all?".

On the other hand, the Ph.D. experience provided me with the opportunity to meet and work with amazing people. A dissertation arises from the work of many: their thoughts, comments, discussions, and related work. There are a few that I would like to explicitly mention here.

First, of course, my supervisor Prof. Dr. John-Jules Meyer. He took me into his group and gave me the opportunity to kick off my Ph.D. program. During the process, he exercised endless patience and provided valuable scientific and personal advices. Without him, I would not have even started it.

Moreover, I am forever indebted to my co-supervisor Dr. Frank Dignum for his incredible support. Frank helped me to (find and) stay on track from the very beginning. His guidance shaped this work. In addition, he is a truly inspiring example of someone that can balance professionalism, patience, encouragement and humanity.

Visiting the University of Melbourne was an incredible experience, both academically and personally. I am grateful to my co-supervisor Prof. Dr. Liz Sonenberg for receiving me there and supporting with ideas, suggestions and valuable guidance. University of Melbourne left such an impression that I decided to apply for the Australia permanent residency and now I am living in Melbourne. I foresee that my involvement with Liz and Melbourne will last for many years to come.

I would like to express my sincerest gratitude to the research groups in both University of Utrecht and University of Melbourne. I have learned a lot

from interacting with this people. Their constructive criticism and collaboration have been tremendous assets throughout my Ph.D. I would like to name a few important contributors. Special thanks to Daghan Acay, my “mate” at University of Melbourne. He helped me with reviews, moral support and sincere friendship throughout the hard process of completing this thesis. Andrew Koster, today at Universitat Autonoma de Barcelona, to whom I exchanged ideas and co-authorship a paper that had a profund impact on this thesis content. Dr. Iyad Rahwan, now at British University in Dubai, a great friend in the early days at University of Melbourne. Along with Dr. Connor Graham we co-authored a few papers that made a great contribution to my research. Last, but not the least, Dr. Virginia Dignum, from University of Utrecht, who helped me on both personal and academic issues.

Finally, to those who were always there with me. Sometimes, small gestures make distinct contributions. I never would make without my family, friends, and loved ones. Thank you!

Chapter 1

Introduction

"The mere formulation of a problem is far more essential than its solution, which may be merely a matter of mathematical or experimental skills. To raise new questions, new possibilities, to regard old problems from a new angle require creative imagination and marks real advances in science."
Albert Einstein.

In this chapter, I define my research question and motivation.

1.1 Brief Background

The dream of intelligent mobile services that assist people during their day-to-day activities in homes, offices, and nursing facilities is compelling. Although this is a favourite subject of software companies, hardware makers, and telecommunication providers, the goal always seems to lie in the future. Norman [1999] suggests that "*following the steps of personal computing, today's mobile services is a frustrating technology; mobile service, as a technology tool, should follow the three axioms of design: simplicity, versatility, and pleasurability*".

As the infrastructure to provide a service, mobile services must be simple, quiet, invisible and unobtrusive. However, counter examples are abundant in everyday life. For example, tour guides disregard the context and promote remote tourism spots, shop assistants deliver inopportune advertisements, and mobile phones ringing during movies and meetings. In these examples, the application does not consider the current situation when delivering the information, becoming inappropriate and inconvenient.

Engineers have yet to solve fundamental problems involving mobile services, such as context-awareness, automated reasoning, pro-activeness, and

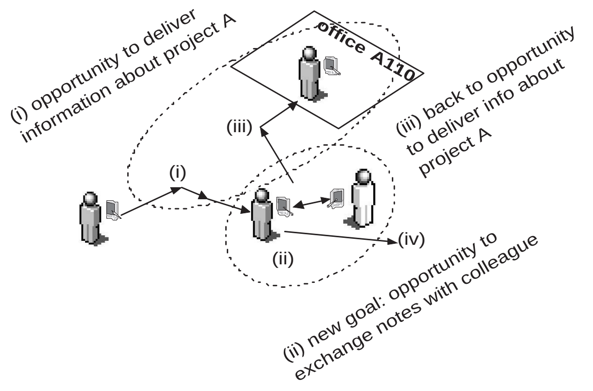

(i) opportunity to deliver information about project A
office A110
(iii) back to opportunity to deliver info about project A
(iii)
(i)
(ii)
(iv)
(ii) new goal: opportunity to exchange notes with colleague

Figure 1.1: Dynamic Environment

adaptiveness. The next generation of mobile services must be invisible, convenient, and useful. It requires new techniques to design and develop mobile computing applications, based on user-centric solutions that are environment-aware and present adaptive behaviour.

In this work, I argue that solutions based on reactive decision loops, such as applications developed upon object-oriented programming languages, are unfit to implement these solutions. As an alternative, I propose a technology based on elements of practical reasoning. I am motivated by the human ability to observe relevant changes of the environment and adapt their line of thinking, based on some innate reasoning mechanism that is both efficient and coherent. This structure requires the integration of context-awareness and agent-based computing technologies. It inherently provides the elements for modelling the user and the environment, and enhanced reasoning.

In what follows, I motivate my research by analysing the requirements to provision intelligent mobile services. Next, I introduce my research question, describe the expected results and present this thesis' structure.

1.2 Motivation

Consider the problem scenario depicted in Figure 1.1 and described below:

Example 1. *A user has a mobile service that helps him with task and time management activities. The application is designed upon the* delegative *model of interaction between the user and the application, where the user decides what to do and which tasks he feels comfortable allocating to the system. It must operate in a fairly autonomous manner with regards to the user behaviour.*

Assume a meeting about Project A is booked for 10am at office A110. At 9:53am, when the user is at a few minutes from office A110, the application starts to retrieve information about the project, intending to de-brief him before he steps into the meeting room (position (i)). When the user detours to a sideline meeting with a colleague (position (ii)), it enters in the window of opportunity to exchange relevant information about the project. Depending on how this conversation evolves, the user might decide to cancel his participation and/or the de-briefing report could extensive change. The application must reconsider its previous processing in order to deliver coherent information.

To fulfil this scenario, the application must be able to sense and represent the environment and user's preferences. In addition, it must adapt its behaviour as the environment evolves. Otherwise, it may incur in incoherent behaviour such as delivering the de-brieing report either not considering the new information from the sideline meeting or out of the opportunity to receive it – e.g. if the user has decided no longer participate in the meeting.

Maes [1994] suggests that "intelligent" software assistants must implement user-friendly interfaces that hide the complexity of the underlying computational process from the user. That is, the application must be able to collect, represent, and process the required information by itself. The less input the application requests from the user, the smarter it is perceived to be, therefore, the first requirement is an ability to interact with the elements of the environment, be autonomous, and situated.

Sheridan [1998] proposes that from the user's point of view an intelligent service must support and take decisions on his behalf. Therefore, the application must be able to start the action by itself, in response to internal and external conditions. In addition, the application must operate for a purpose – such as to support the user with information, task and time management – regardless of the environment's instabilities. This ensures coherence of behaviour.

Moreover, Satyanarayanan [1996] summarises the fundamental issues of mobile environments as:

- *constantly changing context*, as the user moves, engages in new activities, or conditions in the operating environment get updated;
- *spontaneous interactions*, as unexpected situations can interrupt the current processing line;
- *instabilities in the environment*, as mobile communication is highly variable in performance and reliability, and;

- *constraint computing resources*, such as reduced processing capabilities, battery operated, and slow communication channels.

The supporting technology must provide solutions to these issues. Applications built upon it must act coherently by reacting properly to unexpected events, and; making use of environmental information to adjust the processing line. Hence, it *must*:

1. *support to interactions*, i.e. the application must implement the elements to enable it to operate as autonomously as possible; it includes the structures to: (i) collect information from the internal and external environment; (ii) represent this information, and; (iii) process it accordingly;

2. *deliver quality information*, i.e. the application must implement a flexible, adaptive inference system capable of adjusting the processing line in response to internal and external conditions, and;

3. *be resource aware*, i.e. the implementation must be aware of the device's limitations, thus optimised to reduce the amount of computation required to provide the above.

Notice that solutions based on reactive decision loops, such as applications developed in common programming languages like C++ and Java, are unfit to operate in highly dynamic environments. As described in Kinny and Georgeff [1991] pure reactive systems are capable of effective behaviour in some dynamic environments, but cannot guarantee the performance in unanticipated situations. That work suggests that real-time reasoning systems can behave more robustly in unanticipated situations because of their ability to reason about multiple, conflicting goals, their relative value and urgency, and how best to achieve them.

While looking for alternative technologies to support these requirements, I found strong points from the fields of context-awareness and agent-based computing.

Context-aware computing, introduced in Schilit et al. [1994] and Dey [2000], supports representing and reasoning about a current situation. This technology provides partial support to the requirements to (i) *support application's interactions* and (ii) *provide quality information.* However, by itself, it lacks the structures to support pro-active and goal-oriented behaviour. I detail this situation in the next chapter.

On the other hand, *agent-based computing*, presented in Wooldridge and Jennings [1995] and Jennings et al. [1998], provides solutions for: knowledge representation; enhanced inference system; responsiveness and adaptivity; sociability, and; local interactions. In specific, *intelligent software agent* is a sub-class of agent-based computing with the following features:

- *responsiveness*, being able to perceive the environment and respond in a timely fashion;
- *proactiveness*, exhibiting goal-directed behaviour and take initiative when appropriate;
- *sociability*, being able to interact with other agents or humans when needed.

This technology contributes to context-awareness by providing the structures to implement higher-level context information processors. *Vice-versa*, context-awareness technologies contribute to agents by providing the structures to collect, represent, and process environmental information. Combining the two technologies encourages support for intelligent mobile services.

This work is concerned with build application using a specific class of agents: the *Belief-Desire-Intention model.* This model encompasses a respectable philosophical model of human practical reasoning, originally developed by Michael Bratman (*vide* [Bratman, 1987]). It inherently provides the structures to represent the state of the environment (beliefs), outline application's behaviour (desires), represent the current goals (intentions), and describe the rules on how to achieve these goals (plans).

While analysing the support by agent-based technology for applications that execute in highly dynamic environments, I identified shortcomings with regards to the balance between pro-active and reactive behaviour. That is, the existing theories and implementations lack the level of reactiveness required to support these solutions Although the initial designs were inspired by reactive approaches, the focus of the technology evolution was towards pro-active systems equipped to provide adaptive behaviour. This deficiency impacts the support to the requirements for (ii) *delivering quality information* and (iii) *resource awareness* (see above). This assessment motivated me to research for alternative solutions based on the integration and extension of existing theories and technologies.

1.3 Proposed Solution

Agent-based computing inherently supports the requirement to (i) *support to interactions* by providing the structures to:

- represent the environment, as proposed in Weiss [1999];
- respond and adapt the deliberation behaviour, as proposed in Jennings and Wooldridge [1998], and;
- implement sociability and locality of interaction, where agents are able to interact with other agents or humans when needed, as proposed in [de Boer et al., 2003].

In addition, the agent paradigm provides features to implement applications able to operate in dynamic, unpredictable environments, backing the requirement to (ii) *deliver quality information.* In this scope, it provides three desirable features:

1. *Situatedness*, which is the ability to represent and process information about the surrounding environment in order to provide sophisticated support.
2. *Adaptivity*, which is the ability to adjust the deliberative behaviour in response to changes of the environment, offering abstraction tools and computational methods for building software capable of handling unexpected events.
3. *Proactiveness*, which can potentially help build systems that reason about the user's goals and how they may be achieved, supporting coherence.

However, the problem of implementing adaptiveness is to decide *when to adapt*? For instance, let us refer back to example where the user needs to plan for the 10am meeting. The application has a goal to start to retrieve the information when the user is in the window of opportunity to receive it, i.e. near the meeting venue. Once the user steps *in* that area, the application generates an intention to reach that goal and starts to execute it. In addition, the application has a goal to pause that operation if the user moves outside that area. The question is: *when should the application analyse the reconsideration condition*?

Kinny and Georgeff [1991] suggests the following variations of agents configurations, based on how they implement the reconsideration process:

- *bold agents* never consider the impact of environmental changes on the current processing, and;
- *cautious agents* consider the impact at each deliberation step.

Thus, both configurations are inadequate to support the proposed requirements. The *bold agent* does not consider the fact that the user stepped out of the window of opportunity until the current processing completes, which means that it will deliver the notification regardless. Hence, it fails to (ii) *provide quality information.*

On the other hand, the *cautious agent* is continuously analysing the reconsideration condition. Although it succeeds in (ii) *providing quality information*, it does so at the expense of computational resources even when there are no relevant environmental changes to consider. Consequently, it fails to support (iii) *resource awareness.*

Therefore, the solution must be a configuration between these two approaches. I introduce an extended computational model that takes advantage of environmental events to coordinate the deliberation process. It allows the deliberation process to be adjusted in response to impromptu events, and is supported by a context-observer module, based on techniques from context-awareness computing. This module "observes" information about the windows of opportunity to enact the reconsideration process. This architecture delivers a complete solution for the three aforementioned requirements. I provide a proof-of-concept implementation in Chapter 6.

1.4 Research Aims

Based on the analysis above, I claim that current agent platforms cannot be used "as is" to develop intelligent mobile services. I propose to integrate techniques from context-awareness and agent-based computing for this development. My problem statement can be summarised as below:

> What is the architecture of the inference system needed to support
> the development of intelligent mobile services?

Throughout this research, I explore different facets of this question. I introduce an architecture that integrates the elements needed to observe changes of the environment and adjust the processing in response to these events. There are several fundamental issues related to this question, such as:

- *How to implement adaptability in the deliberation process*? That is, how to implement applications whose processing can be adapted on-the-fly, reacting to changes of the environment? This feature is required to support dynamic environments; enabling the application to cope with time-constrained reasoning, unexpected events, spontaneous interactions and other instabilities.

- *How to establish standards of relevance and prioritisation processes*? That is, when the deliberation cycle has multiple alternatives how to determine which one is more suitable at a given time?

- *How to keep coherence and consistency*? That is, how to reason about multiple, conflicting options and sustain means-end coherence? I propose that plan resolution and execution must be consistent with beliefs and rooted in a pragmatic rationale (i.e. goals and norms). It also relates to conflict resolution, as the user might have a number of goals that cannot be achieved simultaneously.

These questions led me to look for alternative solutions, leading me to a new model of agent-based computing deliberation, which is introduced in this work.

1.4.1 Expected Outcomes

My research is concerned with the study of the supporting logic required for designing an extended computational model to support intelligent mobile services. I seek to describe an architecture that enables the processing of simultaneous intentions, and how environmental changes impact their processing.

Applications can take advantage of environmental events to enhance their deliberation performance. That is, events associated with changes of the environment provide the means to decide *when* to reconsider the current intentions. Conversely, extended structures of the deliberation cycle define *how* to implement the reconsideration process. My study focuses on *what* is required to support this integration.

The disciplines that inspire this research range from philosophy to artificial intelligence and software engineering. The topic of autonomous agents has a history based on the philosophy of intention, planning and practical reasoning. The works of Bratman [1987] and Cohen and Levesque [1990] provide the fundamental philosophical concepts of the BDI-model.

My research has been conducted both in the realm of *computer sciences* and *information science*. From the former, I focus on technological aspects of the solution, such as application design, accommodation of the existing technologies, compatibility, optimisations, and so on. From the later, I explore how information about the environment can be represented and used by the application.

I provide a number of illustrative examples throughout the text and case study scenarios in Chapter 6. Ultimately, I expect that the proposed model will provide solutions to real world problems, such as mobility, unexpected events, multiple goals, location sensitiveness, and mobile information service delivery.

1.4.2 Scope

The scope of this research is on software development. Issues regarding market technologies and business models are not pertinent – although I take existing technologies in consideration while implementing the pilot application. I am not concerned with modelling the way humans interact with mobile devices, nor is my work intended to contribute to hardware development. I acknowledge the importance of these areas for providing mobile services, nonetheless.

1.5 Thesis Structure

This thesis is structured in three parts. The motivation is presented in Part I, which details the problem scenario. Part II presents the proposal for the extended logic and design, and Part III validates the approach through case studies. The chapters are distributed as follows:

Part I: MOTIVATION

- **Chapter 2**, *Background and Related Work*, details the requirements to provide intelligent mobile services and categorises solutions offered by the related work.

Part II: PROPOSAL

- **Chapter 3**, *Conceptual Model*, introduces the concepts and provides a clear picture on *why* agent applications must be equipped to adapt their deliberation cycles, and *what* are the structures required to support adaptation.
- **Chapter 4**, *Design Model*, extends the definitions proposed in the previous chapter and introduces the design model that explains *how* to design intelligent mobile services.

Part III: SOLUTION

- **Chapter 5**, *Prototype*, discusses the implementation of a pilot application, aiming to demonstrate that the specifications are comprehensive and reproducible in software agent applications.

- **Chapter 6**, *Case Studies*, presents a set of qualitative and quantitative analysis of case study scenarios.

Finally, **Chapter 7**, *Conclusion*, summarises the progress achieved in the body of the thesis, highlights its contribution, and suggests the path for future research.

Chapter 2

Background and Related Work

"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."
Mark Weiser.

In this chapter, I analyse the problem of providing intelligent mobile services.

2.1 Introduction

This chapter introduces an analysis of what is required to implement intelligent mobile services. It also provides an overview of the related techniques and theories, aiming to distil the fundamental concepts behind the problem scenario. In summary, the goal of this exercise is to:

- identify the requirements to provision intelligent mobile services;
- categorise the solutions provided by the related work;
- investigate their shortcomings, and;
- describe how the solution being proposed in this work is related and/or extends the related work, covering their gaps.

The chapter is organised in four sections. The first section introduces the requirement analysis and review the essential features of the problem scenario. Section 2.3 analyses the techniques and theories provided by context-aware computing. Section 2.4 analyses the solutions provided by agent-based computing. Section 2.5 concludes with a discussion of the proposed work.

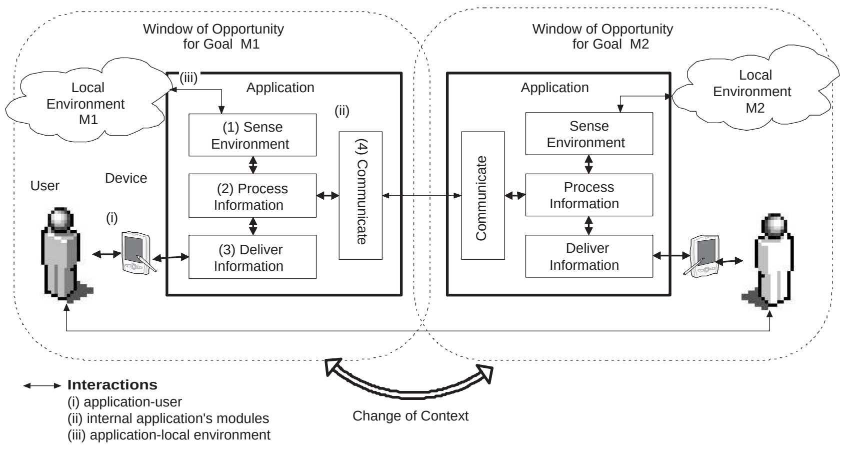


Figure 2.1: Elements and Interactions in Mobile Services

2.2 Background

This section identifies the requirements for providing intelligent mobile services by analysing the elements and interactions of the problem scenario.

2.2.1 Elements and Interactions

Let us recall the problem scenario from the previous chapter. The application has to provide information related to a goal $M1$ while the user is in the window of opportunity for that goal. However, at a given point, an unexpected event happens and the user changes its course, moving towards the goal $M2$. At this point the context changes as the user leaves the window of opportunity for $M1$ and enters the area for $M2$. The application must sense the new situation and adapt its processing, refraining from delivering information about $M1$ when outside its window of opportunity.

In order to implement its basic functionality, mobile applications must implement at least four sets of structures as it is intuitive and depicted in Figure 2.1:

1. to sense the environment;
2. to process external and internal information;
3. to deliver information, and;
4. to communicate with other applications.

The problem of implementing these applications is related to the question: *how to integrate these components so that they behave in the right way*? It is expected that mobile services implement location-based, decision support information systems with adaptive and pro-active behaviour in order to display "intelligent" behaviour. The application must be able to do all that and execute in constraint resources, which is an inexorable fact of the environment, as proposed by Satyanarayanan [1996]. In this work, I am proposing to integrate technologies and theories from different fields of research to reach this ideal.

I analyse the interactions between these components in order to clarify the scope of this study. Figure 2.1 depicts three groups of relevant interactions, described below:

- (i) *interactions between application and users* describe how the user interfaces with the application. The information provided to the user is only as good as the quality of the processing. Hence, this interaction is strongly influenced by the other elements of the application. On a lower-level of abstraction, it involves issues of human-computer interaction (which are outside the scope of this work).

- (ii) *inter-application interactions* describe the relationship between the application's modules. This is related to how the application is programmed and how it interacts to the external world. The application model includes four functionality groups: (1) modules to sense the environment; (2) modules to process information; (3) module to deliver information to the user, and; (4) modules to promote inter-application communication. This work focus on improving the first two groups.

- (iii) *interactions between the application and the local environment*, describe the interactions to external elements. Besides the (1) module to sense the environment, the application must also implement structures to represent and reason upon this information. These interactions exist in the realm of context-aware computing, which provides the techniques to collect, represent, and infer on environmental information.

In the following sub-sections, I analyse the requirements to support the three pertinent groups of interactions introduced above.

2.2.2 Supporting Enhanced User Experience

User experience is a key focus on services development, dealing with aspects of the user's interactions with the product: how it is perceived, learned, and used. The primary emphasis is on the usage phases of the product. It covers technical aspects, such as developing a conceptual model for the user, as well as experimental, social, and behavioural sciences.

Myers et al. [2007] discusses a solution where the application implements time and task management functionalities. That work proposes four requirements for enhance user experience, which I adopt as the *requirements* for this group of interactions:

- (i) *directability* allows the user to guide the strategy used in solving a particular task, and the scope of the system's autonomy;
- (ii) *personalisation* equips the application to adapt to the user's preferences and characteristics over a wide range of operations, including how tasks are performed, and the system interacts with the user;
- (iii) *teachability* enables procedure knowledge to expand and modify at run time; an essential feature for a persistent problem-solving agent that is supposed to operate autonomously, and;
- (iv) *transparency* equips the application to explain how decisions are made, so users are confident about the quality and thoughtfulness of the application.

Moreover, the application (v) *must be able to display pro-active behaviour. Proactiveness* is the ability to prescribe plans that deal with foreseeable or unfolding situations. Myers et al. [2007] proposes that proactiveness "*increases the overall effectiveness of personal assistants*". An application that presents proactive behaviour is able to take actions considering the possibility of future events. Therefore, it can work on behalf of the user without requiring a direct command to execute an action. For example, by preparing background material for the meeting without being explicitly asked.

Isbell and Pierce [2005] proposes that the level of proactiveness in a personal assistant ranges from zero to full automation in the *interface-proactivity continuum.* That is, the application can be classified between: "Do It Yourself", "Tells You to Pay Attention", "Tells You What to Pay Attention To", "Makes Suggestions", and "Takes Decisions". The proposed design must provide the elements to support any of these levels, depending on the features of the solution.

This work aims to provision mobile services that support these requirements and implement, at least, "Make Suggestions" level of automation.

2.2.3 Supporting Inter-Application Interactions

Perry et al. [2001] describes that mobile services must be able to: sense the environment; create a representation of the world; represent the user's preferences; reason about alternative plans, and; deliver the right information at the right time. Therefore, the three core elements of the application architecture – i.e. the modules to (1) sense environment, (2) process information, and (3) deliver information – must act accordingly. I suggest that the application must implement the following requirements to support these features:

- (vi) it *must be* able to adapt its processing in reaction to environmental events. This is related to the issue of responsiveness and adaptability in software applications. It requires the application to be able to represent events of the environment and reason upon this information to produce decisions;
- (vii) it *must be* able to sustain means-end coherence throughout the planning process, thus keeping consistency and coherence. That is, the application must work autonomously and behave flexibly in reaction to unexpected situations, and;
- (viii) it *must be* able to adjust the deliberative behaviour in response to changes of the environment. That is, it must be able to learn from the environment and experience and be equipped to adjust the deliberation behaviour in the short-term, and also fine tune the operational parameters to ensure the performance gains in the long term.

Although it is possible to conceive reactive-decision loops that implement these functionalities *ad hoc*, I claim the need for a generic solution based on modules that inherently provide these features. This design allows the application to exploit features of the windows of opportunity to improve the deliberation behaviour. It contributes to support the requirements above and towards the development of intelligent mobile services.

2.2.4 Supporting Elements of the Environment

There is also the requirement to support interactions between the application and the local environment. Huang et al. [1999] proposes that "*the power of pervasive computing is unleashed when the application has the intelligence to process contextual information about the user and the environment in order to provide the user with the right information at the right time*". For example, applications equipped with this functionality do not need to ask "where are we?" or "what objects are available?". Instead, they can collect this information from the surrounding environment *via* an appropriate sensorial system.

Mobile service environments present some distinct features, as summarised below:

- *constantly changing context*, which is related to mobility; the user engages in new activities, or conditions of the operating environment need to be updated;
- *spontaneous interactions*, abrupt changes in computational priorities due to unexpected situations, and;
- *environmental instabilities*, which is a common issue in mobile services related to transient tasks, fluctuations in connectivity, and other operational issues.

In order to design applications capable of operating autonomously and/or reactively in this scenario, I highlight the concepts and technology provided by *context-aware computing*. This research field provides the means to capture, represent and process context information, allowing the application to respond to the surrounding environment. Schilit et al. [1994] explains the rationale to explore the context-awareness in mobile computing as:

> "*One challenge of mobile distributed computing is to exploit the changing environment with a new class of applications that are aware of the context in which they are run. Such context-aware software adapts according to the location of use, the collection of nearby people, hosts, and accessible devices, as well as to changes to such things over time. A system with these capabilities can examine the computing environment and react to changes to the environment. Three important aspects of context are: where you are, who you are with, and what resources are nearby. Context encompasses more than just the user's location, because other things of interest are also mobile and changing. Context includes lighting, noise level, network connectivity, communication costs, communication bandwidth, and even the social situation; e.g., whether you are with your manager or with a co-worker.*"

Dey et al. [2001] proposes that the framework to design context-aware applications requires: (i) the means to collect information from the environment, i.e. the sensors; (ii) the structures to represent context information, and; the (iii) structures to process context information.

There are several dimensions of context information that can be applied. Graham and Kjeldskov [2003] proposes eight segments of context information as: time, absolute location, relative location, objects present, activity, social setting, environment, and culture. Hong and Landay [2001] have described context as knowing the answers to the "W" questions, such as "Who is speaking", which relates to the Graham's descriptions as: When? (time); What position? (absolute location); Where? (relative location); What else? (objects present); What work? (activity); Who? (social setting); What condition? (environment); and What culture? (culture). It is possible to conceive applications in a variety of knowledge domains by exploiting context-awareness in these different segments.

In order to support context-awareness, I introduce the following *requirements* for the application's design:

- (ix) it *must provide* the structures for knowledge representation to provide an explicit representation of the environment;
- (x) it *must be* situated in the environment; that is, be able to represent and process information about the surrounding environment in order to provide sophisticated support, and;
- (xi) it *must be* scalable and equipped to support large amounts of information and to cross-relate a number of environmental information, user's preferences, available resources, possible conflicting goals. That is, it *must be* designed to optimise processing by rationalising the environmental information it has to process.

Next, I analyse the role of the related technologies and theories in supporting these requirements.

2.2.5 Analysis of the Related Technologies

Table 2.1 analyses the role of context-awareness and agent computing technologies in supporting the requirements aforementioned. The following paragraphs discuss the details.

The Role of Context-Awareness Technology

The role of context-awareness technology is to provide structures that capture, represent and process contextual information. They support the requirements as analysed in Table 2.1 and detailed below.

The (i) sensors are the ingress point for contextual information. They provide data input to the other structures, and are essential to support the requirements for *responsiveness* and *situatedness*. Moreover,it is possible to extend the sensor algorithm to filter relevant environmental updates, thus saving resources and contributing to *scalability*.

	Context Awareness			Agent Technologies				
Technique Requirement	Sense Environment	Represent Context Information	Process Context Information	Structures Knowledge Representation	Enhanced Inference Systems	Responsiveness and Adaptability	Sociability and Local Interactions	Autonomy
(i) Directability				Simple structures to represent knowledge to be communicated	Decision process can be explained based on deliberation steps		Supports communication to user	
(ii) Personalisation		Represents user's preferences		Represents user's preferences	Processes based on user preferences		Supports communication to user	
(iii) Teachability				Simple representation of processing rules	Allows modification of processing rules		Supports communication to user	
(iv) Transparency				Simple representation of processing rules	Allows analysis of deliberation steps		Supports communication to user	
(v) Pro-Activeness					Supports pro-active processing of existing goals			Allows agent to work by itself
(vi) Responsiveness	Collects context information to allow application respond to it		Allows low-level reactiveness		Supports reactive processing of environmental conditions	Responsiveness to external events		
(vii) Means-end coherence					Processes information based on desired goals			Processes information based on desired goals
(viii) Adjustable and self-configurable				Representation of processing rules allow on-demand configuration	Allows on-demand reconfiguration of processing rules	Allows adjusting of current processing in response to external events		Allows to self-configure
(ix) Structures for knowledge representation		Represent knowledge about the environment		Representation of processing rules				
(x) Situatedness	Collects context information in order to situate the application	Represents current situation	Infers current situation	Represents the current situation	Processing of current situation		Supports interaction to local environment	
(xi) Scalable	Requires the methods to filter relevant context information				Balances reactiveness/ proactiveness for scalability	Reactiveness balances proactiviness helping scalability		

Table 2.1: Support Provided by Context-Aware and Agent Technologies

The (ii) structures to represent contextual information provide the organised storage that contains user's preferences, knowledge about the environment, and knowledge about the situations. This information is essential for supporting the requirements for *personalisation*, *environment representation*, and *situatedness* respectively.

Finally, the (iii) structures to process contextual information allow first level (i.e. reactive) processing of context information. One shortcoming of current implementations of context-aware techniques is that they provide *ad hoc* solutions, based on the requirements of that specific application. Therefore, the problem is not how to sense and collect information from the environment, but rather how to represent and process meaningful context about this information?

Concepts of context-aware computing and agent-based computing can be integrated to create intelligent mobile services. This combination would equip context-aware solutions with a "*sophisticated interpretation* and, equip agent's deliberation cycle with the means to perceive, represent, and interpret contextual information in order to improve its performance. A contribution of this thesis is to implement higher-level context information processors, which is a key feature for producing sophisticated context-aware applications, as described by Dey [2000].

There is a large body of research about context-awareness as detailed in Section 2.3.

The Role of Agent Technology

Agents present a compelling vision of future computational systems by introducing a solution for representing computer systems in terms of multiple interacting autonomous units. Inverno and Luck [2003] describes an *agent* as "*an autonomous computer system that is situated in an environment, and capable of flexible autonomous behaviour in order to meet its design objectives*". An *intelligent agent* is a class of software agents that are: (i) responsive (or adaptive), i.e. able to perceive the environment and respond in a timely fashion; (ii) proactive, i.e. exhibit goal-directed behaviour and take initiative when appropriate, and; (iii) social, i.e. able to interact with other agents or humans when needed.

Maes [1994] proposes the use of agents for the development of personal assistants, as quoted:

> "*Techniques from the field of Artificial Intelligence, in particular so-called autonomous agents, can be used to implement a complementary style of interaction, which has been referred to as indirect*

management. Instead of user-initiated interaction via commands and/or direct manipulation, the user is engaged in a cooperative process in which human and computer agents both initiate communication, monitor events and perform tasks. The metaphor used is that of a personal assistant who is contributing with the user in the same work environment. The assistant becomes gradually more effective as it learns the user's interests, habits and preferences (as well as those of his or her community). Notice that the agent is not necessarily an interface between the computer and the user. In fact, the most successful interface agents are those that do not prohibit the user from taking actions and fulfilling tasks personally. Agents assist users in a range of different ways:

- *they perform tasks on the user's behalf*;
- *they can train or teach the user*;
- *they help different users collaborate*;
- *they monitor events and procedures.*

In Koch and Rahwan [2005], we suggested that *the role of agent technology* in the development of intelligent mobile services is to provide (i) the structures for knowledge representation; (ii) enhanced inference system; (iii) responsiveness and adaptivity, and; (iv) sociability and locality of interaction. Table 2.1 describes these elements and correlate to the requirements to develop mobile services introduced in the previous section. The correlation is detailed below.

First, *structures for knowledge representation* provide the means to support the requirements for situatedness; enhanced user interface; self-adjustment; representation of processing rules, and representation of the situation. Weiss [1999] shows that the agent paradigm has produced a variety of methods for explicit representation of the environment, and for reasoning about this environment to produce decisions.

Second, *enhanced inference system* is a characteristic of current agent implementations. These structures are usually implemented as planning systems, based on cognitive behaviour. This system *animates* the agent by executing the application in response to internal and external conditions, according to planning rules. This feature provides support for the requirements for enhanced user interface; pro-activeness; reactiveness; means-end coherence; self-adjustment, and; situatedness.

Moreover, *responsiveness and adaptivity* are inherent features of agent systems, as pointed out by Jennings and Wooldridge [1998]. Agents must be able to adapt to constantly changing execution environment. This feature provides support to responsiveness to external events and adjustable and self-configurable behaviour, allowing the adjustment of current processing line in response to external events. This is a core functionality to promote the balance of reactiveness and pro-activeness on the deliberation cycle, thus contributing to system scalability.

In addition, *sociability and locality of interaction* are also inherent features of the agent technology. Jennings and Wooldridge [1998] describes that agents are able to interact with other agents or humans when needed; sophisticated interaction mechanisms having been developed in the agent community to facilitate information exchange, coordination, collaboration, and negotiation. For example, see [de Boer et al., 2003], [Durfee, 1999], and [Beer et al., 1999]. These mechanisms support the requirements for enhanced user experience and situatedness.

Finally, agent-technology provides the mechanisms that address varying degrees of *autonomy*, from basic reactive architectures based on a set of pre-determined rules, to mechanisms for proactive behaviour, as described in Dastani et al. [2003]. The application considers context, user preferences, and configured goals to execute autonomously from any user input and environmental conditions. This is a base feature to support pro-activeness and self-adjustment.

In Section 2.4, I qualify the solutions provided by agent technology and analyse their shortcomings, aiming to identify the gap where this work makes its contribution.

2.3 Context-Awareness Technology

In this section, I introduce the related work in context-awareness, and; cross-relate solutions provided by these works with the requirements outlined in the previous section.

The context-awareness paradigm can be applied to any type of solution, such as contextualised web browsing; field operations; office assistants, as described in Yan and Selker [2000]; museum guides, as introduced in Koch and Sonenberg [2004], and; tourism guides, as presented in Abowd et al. [1997], Pashtan et al. [2003]. Baldauf et al. [2007] presents a survey on different design principles and context models for context-aware systems and presented various existent middleware and server-based approaches.

The *MIT's Project Oxygen* [1] intends to deliver an intelligent environment

[1]MIT's Project Oxygen: web-site at http://oxygen.lcs.mit.edu/, accessed in Nov-2008

through the use of context-awareness and invisible computing technologies. The project aims at a human-centred computation where the user will be able to freely access the services. The authors describe the services as "*enter the human world, handling our goals and needs and helping us to do more while doing less*". This visionary project generated new paradigms for software and hardware integration.

Project Endeavour [2], by University of California, Berkeley, is even more visionary. Its goal is to "*specify, design, and prototype a planet-scale, self-organising, and adaptive information utility (...) that distribute itself among Information Devices and along paths through scalable computing platforms integrated with the network infrastructure, compose itself from pre-existing hardware and software components, satisfy its needs for services while advertising the services it can provide to others, while negotiating interfaces with service providers while adapting its own interfaces to meet "contractual" interfaces with components it services.*" However, this project focuses on how to collect-index-represent, leaving aside issues of enhanced processing and pro-activeness required by more elaborated services.

Project Portolano [3], presented in Esler et al. [1999], seeks to create a test-bed for investigating the emerging field of invisible computing. The devices are optimised to particular tasks, such that they blend into the world and require little technical knowledge on the part of their users. This project focuses on issues of sensing, context modelling, and context processing, leaving future work to explore supporting more elaborate services.

Service-Oriented Context-Aware Middleware (SOCAM) project, introduced in Gu et al. [2004], is another architecture for the building and the rapid prototyping of context-aware mobile services. It uses a central server that holds context data through distributed context providers and offers it in mostly processed form to the clients. The context-aware mobile services sit on top of the architecture, making use of the different levels of context and adapting their behaviour according to the current context. Similar to the previous project, this project focuses on solutions for sensing, context model, and resource discovery. It does not directly support enhanced deliberation, although it does provide the means for external elements to interface with the context repositories to access this information.

MyCampus[4], introduced in Sadeh et al. [2003], targets context-aware mo-

[2]Project Endeavour: web-site at http://endeavour.cs.berkeley.edu/, accessed in Nov-2008

[3]Project Portolano: web-site at http://portolano.cs.washington.edu/, accessed in Nov-2008

[4]Project MyCampus: web-site at http://www.cs.cmu.edu/ sadeh/mycampus.htm, accessed in Nov-2008

bile services. The environment revolves around a growing collection of task-specific agents capable of automatically accessing a variety of contextual information about their users, for example context-aware restaurant concierges and filtering agents. The work in *MyCampus* focuses on technology development, such as the OWL Semantic Web reasoning engine, context-aware agents, location tracking functionality, and OWL Rule Extension, and; human-computer interface evaluation methodologies. Again, it does not provide a clear specification for the development of the intelligent applications.

These projects focus on the technologies for context sensing and representation, which is largely the case of current research in context-awareness technologies. These concepts can be re-used as they provide the lower-level functionality to the model that I am proposing in this work. However, they need to be extended with features of enhanced deliberation, means-end coherence, and situatedness to support intelligent mobile services. Next, I explain the integration to agent-based technology.

2.3.1 Context-Awareness and Agent Technology

The idea of integrating context-awareness technologies to agent computing is not new. Sahli [2008] introduces a survey of agent-based middleware for context-awareness, analysing the current developments from the perspectives of implementation, reasoning, negotiation, and scalability. The author concludes that:

> "*Agents seem to be very suitable to fulfil the requirements of modularity, scalability, adaptability, and distributedness of middleware for context-awareness. However, the achievement of all these requirements will depend on the use of the agent paradigm as a central paradigm in the considered systems.*"

In this work, I am looking for generic technologies and concepts that can be adapted and re-used towards the development of mobile services. Hence, I analyse the projects that already promote integrating context-awareness and agent technology, and introduce a comparative analysis of *capabilities versus requirements* for a number of these projects in Table 2.2.

Electric Elves Project, introduced in [Chalupsky et al., 2001], exploits agent technology to assist users with daily tasks, such as organising meetings. The project includes the notion of context-awareness, personal assistance, and collaboration. Tambe et al. [2008] analyses "what went wrong and why" with regards to the *E-Elves project*. In a paper outlining some of the lessons learned

	Context Awareness Projects				
Project / Requirement	**Electric Elves**	**CyberGuide**	**AMUSE**	**Stu21**	**ACAI**
Class of Application	Task management	Tour Guide	Ubiquitous Environment	Smart spaces and educational institutions	Framework for Agent Solution
(i) Directability	User can choose task decision rule		User can set operational parameters		
(ii) Personalisation	Provides personalised information	Personalisation based on user's interests	Personalisation of Users' Preferences and Operational Parameters		
(iii) Teachability	Provides interface to introduce new task management rules	Not clearly available	Not clearly available	Customisable rules engine	
(iv) Transparency	Provides information on decision taking process	Provides information on decision taking process	Provides some information on decision process	Provides information on decision taking process	
(v) Pro-Activeness					
(vi) Responsiveness	Related to environmental events	Related to detected location	Responsive to environment and operational parameters	Able to initiate, execute, and terminate behaviours in response to context change	
(vii) Means-end coherence				Partial: rule-based deliberation system with representation of beliefs and goals	
(viii) Adjustable and self-configurable	Provides extension for feedback process			Provides learning capabilities	
(ix) Structures for knowledge representation	Representation of environment and user's preferences	Representation of environment and user's preferences	Representation of environment and operational parameters	Able to keep a model of context	Extensive discussion on knowledge representation and context reasoning
(x) Situatedness	Partial: situated in the environment and tasks' status	Partial: related to location and user's preferences	Not clearly available	Represent local environment and situation	Propose methods to represent local environment
(xi) Scalable					Proposes a multi-agent framework that parallels the infrastructure design
Re-Usable	Concepts are re-usable	Concepts are re-usable	Concepts are re-usable	Several concepts can be re-used	Several concepts on knowledge representation and context reasoning can be re-used

Table 2.2: Context-Awareness' Implementations *versus* Requirements

from a successfully-deployed team of personal assistant in an office environment, the authors identified the shortcoming that agents must obey the social norms in order to operate successfully. *E-Elves* covers several points of the requirements for intelligent mobile services being proposed in this work, however, this development is not based on common, re-usable agent technologies. Hence, although several of the concepts can be re-used, it is difficult to apply the specifications in other domains of knowledge.

Project CyberGuide, introduced in [Abowd et al., 1997], implements, validates, and trials a tourism-related value-added service for nomadic users. This implementation can be classified as a mobile personal assistant, although the focus is on a specialised service for tourism guide. The project is based on agent technologies, thus some of the concepts in information gathering and knowledge representation can be re-used in the scope of this work. The project does not exploit the issues of enhanced deliberation and pro-activeness, however, leaving a gap and opportunity for this current work to contribute.

AMUSE, presented in Takahashi et al. [2005], proposes an agent-based middleware for ubiquitous environments comprised of computers and home electric appliances. Moreover, it embeds long-term context-maintenance ability to the agents to accumulate cooperation history and experiences. AMUSE targets interfacing with human users and a process to take quality of service (QoS) in ubiquitous environment. Some of the QoS concepts regarding processing reconsideration and reactiveness to established operations parameters can be re-used, however, it does not addresses the intelligent mobile services issues of personal assistance, pro-activeness, etc.

STU21 project, introduced in Singh and Conway [2006], investigates how individuals may interact with smart spaces through the concept of a "context browser". In this project, the context browser is regarded as an update to the standard web browser that dynamically changes as context changes. The browser allows an individual to interact with information and services based on context. It has been ported to mobile devices, creating a portable interface to the world as seen through the various peer agents. *Stu21* shows a good example of how an agent can be aware of the context. The agent is able to keep a model of context, process it, deduce beliefs and goals, communicate changes to other agents, and initiate, execute, and terminate behaviours in response to context change. Several concepts can be re-used, such as the *context observer*, the interaction between this module and the rule-based reasoning system, and the inter-agent actions.

ACAI Framework, presented in Khedr and Karmouch [2005], proposes agents that accomplish several requirements of context-awareness, for example context composition, inferring, and inter-domain context invocation. The goal is to provide an infrastructure that can understand the current situation and act on that understanding. The infrastructure supports uniform context representation, reason about context, context-based service discovery, and context management and communication protocol. The solution is equipped with the capabilities required to maintain spontaneous applications, both locally and across different domains. This project focuses on issues of context representation and reasoning in agent systems. Several concepts on knowledge representation and context reasoning can be re-used; however, the project leaves open issues of pro-activeness, application stability and means-end coherence, which are required for intelligent mobile services.

2.3.2 Analysis and Shortcomings

It is clear that the existing solutions focus on *ad hoc* implementations for specific problems. The focus of these developments is on techniques to collect, index and represent contextual information! Baldauf et al. [2007] suggests that a future work is "*to build more sophisticated algorithms, which derive new contextual knowledge or patterns to proactively aggregate new context-aware services*". Hence, these solutions are not designed to support pro-active, situated systems that are able to implement variable levels of automation. This shortcoming can be amended by elements of agent technologies, which inherently provide enhanced deliberation systems, pro-active behaviour, and situatedness.

Integrating context-awareness and enhanced inference systems can overcome the imbalance between reactiveness and proactiveness in current AI systems. The model must be defined in such a way that it not only observes the context change but also knows how to adapt the deliberation behaviour accordingly.

In this thesis, I am extending the BDI-model of agent computing with the introduction of a *context observer module*. This module works by rationalising relevant changes of the environment; a concept derived from the *STU21 project*. Throughout this work, I demonstrate *what* is this structure and *how* it helps to optimise the agent's deliberation performance. Moreover, I am proposing to extend the planning module with simultaneous deliberations and deliberation checkpointing – i.e. the ability to pause, resume, and stop individual deliberation threads. This development covers the gaps of existing on the analysed implementation.

Next, I discuss the related work on agent technology.

2.4 Agent Technology

Agent technology plays a key role in implementing intelligent mobile services. This solution complements the features provided by context-awareness technologies, supporting enhanced user interface and inference systems, pro-activeness, responsiveness, means-end coherence, adjustability, situatedness, and scalability.

In this sub-section, I introduce the related work in agent-based computing. I focus on the three main concepts implemented by this technology:

- *cognitive behaviour*, which proposes computational processes that act like certain cognitive systems present in human beings; providing a solution for means-end coherence; adaptability, and pro-active behaviour;

- *planning systems*, which provides a solution for enhanced inference system, and contributes to pro-active behaviour, enhanced user interface, such as directability and transparency, and means-end coherence;

- *practical reasoning*, which provides a solution for extended knowledge representation; teachability, personalisation, means-end coherence, and responsiveness; achieved by exploiting concepts of BDI-model of agency.

Table 2.3 describes the support of example implementations of these conceptual models to the requirements of intelligent services. It is clear that these features help to complete the deficiencies of context-aware computing, outlined above. I introduce the related work, qualify the solutions, and analyse their shortcomings in the following sub-sections.

2.4.1 Cognitive Behaviour

Cognitive behaviour proposes computational processes that act like certain cognitive systems present in human beings. It intends to simulate “intelligent behaviour” under some definition. [Lehman et al., 2006] proposes that this behaviour is the solution to overcome the weaknesses of current software development processes, based on reactive decision loops. It would result in intelligent behaviour through cognitive architectures, which form a subset of general agent architectures. I highlight that the term “architecture” implies to model not only behaviour, but also structural properties of human deliberation. This concept develops into models for practical reasoning, such as *Belief-Desire-Intention model*, which I elaborate later in this chapter.

	Cognitive Behaviour			**Planning**	**Practical Reasoning**	
Architecture / Requirement	**SOAR**	**CLARION**	**ACT-R**	**STRIPS**	**PRS**	**BDI-Model**
Type of Solution	Production System	Cognitive Architecture	Cognitive Architecture	Planning System	Agent Architecture	Agent Architecture
(i) Directability	Possible to incorporate interface so use can choose the group of rules to use	Possible to incorporate, but not available	Possible to incorporate, but not available		Decision process can be explained based on deliberation steps	Decision process can be explained based on deliberation steps
(ii) Personalisation			Available through User modelling			Models user's preferences and behaviour
(iii) Teachability	Reconfiguration of decision taking rules	Reconfiguration of decision taking rules	Reconfiguration of decision taking rules		Possible to assert new knowledge on-demand and at run-time	Possible to assert new knowledge on-demand and at run-time
(iv) Transparency	Provides information on executed rules for decision taking	Provides information on executed rules for decision taking	Provides information on executed LISP rules	Provides information on executed rules for decision taking	Provides information on executed rules for decision taking	Provides information on executed rules for decision taking
(v) Pro-Activeness	Partial: supported by means-end coherence	Partial: supported by means-end coherence	Partial: supported by means-end coherence	Partial: supported by means-end coherence	Supported by goal oriented behaviour	Supported by goal oriented behaviour
(vi) Responsiveness		Implements serial reaction time and process control tasks			Enhanced responsiveness by allowing reconsideration of environmental situation in between plan steps	Enhanced responsiveness by allowing reconsideration of environmental situation in between plan steps
(vii) Means-end coherence	Search through a problem space for a goal state	Search through a problem space for a goal state	Search through a problem space for a goal state	Search through a problem space for a goal state	Goal-oriented behaviour and desire-belief plan selection	Goal-oriented behaviour and desire-belief plan selection
(viii) Adjustable and self-configurable	Possible to adjust production rules at running time	Provides bottom-up and independent rule learning	Focus on learning by exploration and demonstration		Allows adjusting of current processing response to external in events	Allows adjusting of current processing response to external in events
(ix) Structures for knowledge representation	Able to create representations and use appropriate forms of knowledge	Provides structures for symbolic information representation	Represent as symbols in Lisp structures	Provides structures to represent environment	Representations of goals, plan rules, beliefs	Beliefs, Desires, Intentions, and Plan Rules
(x) Situatedness	Partial: Provided as representation of search space	Representation of internal and external states	Partial: Related to status of search space		Supports interaction and representation of local environment	Supports interaction and representation of local environment
(xi) Scalable					Simplified planning process	Simplified planning process
Re-Usable	Several concepts are re-usable	Re-use the concept of concept of serial reaction time and process control		Provides Core concepts of planning system' operations	Concepts	Concepts and Implementation

Table 2.3: Agents' Concepts *versus* Requirements

Implementation

I introduce a number of significant implementations in related work, although I qualify the support these implementations provide to the requirements for intelligent mobile services in Table 2.3.

SOAR project, introduced in Rosenbloom et al. [1993], is an example of a cognitive architecture implementation, which evolved from the *General Problem Solver* (GPS), introduced in Newell et al. [1959]. The main goal of the SOAR project is to handle the full range of capabilities of an intelligent agent, from routine to extremely difficult open-ended problems. To implement this functionality, it is able to create representations and use appropriate forms of knowledge – such as procedural, declarative, episodic, and possibly iconic models. SOAR is based on a production system that uses explicit production rules to govern its behaviour. Problem solving can be roughly described as a search through a problem space for a goal state; thus, this project lays the basis for means-end coherence in software applications. The concepts introduced in SOAR provide support to a number of the requirements for intelligent mobile services, such as teachability, transparency, means-end coherence, adjustability, and structures for knowledge representation.

Connectionist Learning with Adaptive Rule Induction ON-line (CLARION), presented in Sun [2008], is a cognitive architecture that incorporates the distinction between implicit and explicit processes. CLARION has been used to simulate several tasks in cognitive psychology and social psychology, and has also been used to implement intelligent systems in artificial intelligence applications. CLARION is an integrative architecture, consisting of a number of distinct subsystems, with a dual representational structure in each subsystem (implicit versus explicit representations). Its subsystems include the action-centred subsystem, the non-action-centred subsystem, the motivational subsystem, and the meta-cognitive subsystem. Moreover, it implements serial reaction time and process control tasks and provides high-level cognitive skill acquisition tasks. The concepts introduced in CLARION are aligned to several requirements in intelligent mobile services, such as teachability, means-end coherence, adjustability, and structures for knowledge representation. The concept of serial reaction time and process control provides the base for the reaction process in the architecture proposed in this thesis.

Adaptive Control of Thought–Rational(ACT-R), presented in Anderson and Lebiere [1998], is a cognitive architecture that aims to define the basic and irreducible cognitive and perceptual operations that characterise the human mind. ACT-R theory has a computational implementation as an interpreter of the Lisp language. ACT-R focuses on the issues of learning by exploration and demonstration, skill acquisition, cognitive arithmetic, and memory. These concepts could be applied to enhance the learning mechanism of the proposed model, however, as issues of learning are out of the scope of this thesis, I do not explore this possibility extensively. I highlight the development on *personalisation* for the application of ACT-R in intelligent tutoring systems, as described in Anderson and Gluck [2001]. Concepts from this application can be re-used to support the personalisation requirement.

Of course, there are other architectures and implementations available such as *CHREST*, presented in Gobet and Lane [2004]; DUAL, developed at the New Bulgarian University and presented in Kokinov [1994]; *Psi-Theory*, from Otto-Friedrich University and presented in Bartl and Doerner [1998]. However, I argue that these works make use of techniques that are far from the goals of this thesis, such as teaching cognitive modelling, integration of cognitive and symbolic system, and human action regulation, intention and behaviour, therefore they are less significant in the context of this work.

Analysis and Shortcomings

I summarise the support of this model to the requirements of mobile services in Table 2.3. The contribution is multi-fold, as the application is able to:

- *behave in a goal-oriented manner*, as cognitive architectures implement rule-based reasoning systems to deliberate on how to achieve given goals and adapts the execution path in response to changes of the environment towards that objective; resulting in more stable, more coherent applications;
- *operate in a rich, complex, detailed environment*, due to rule-based behaviour and operations toward goal resolution, which resembles means-end coherence;
- *use symbols and abstractions*, as cognitive architectures represent the environment by *labelling* its elements;
- *learn from the environment and experience*, as cognitive architectures are able to adjust the deliberation behaviour in the short-term and also fine tune the operational parameters to ensure the performance gains in the long term.

However, implementing cognitive behaviour is not a trivial task, involving resource-consuming, complex operations that are difficult to program. The challenge is how to bring the benefits of the cognitive behaviour concept into constraint resource environments, *viz* mobile computing. Thus, the main shortcoming of this approach is the lack of support for the requirement of *scalability*.

Moreover, these architectures focus on the core functionality of cognitive behaviour and do not provide direct support to requirements of enhanced user-interface – i.e. directability, personalisation, teachability, and transparency. Customising the implementation can deliver some of these requirements. Finally, these architectures do not support pro-active behaviour, expecting pro-activeness as an emergent feature.

In this thesis, I am introducing an extended model that re-use some concepts from these works but also focus on reactiveness to environmental events and scalability, addressing the gaps of these works for the development of intelligent mobile services.

2.4.2 Planning Systems

Planning systems provide a solution to some of the gaps in context-aware and cognitive behaviour technologies, such as enhanced inference system, pro-active behaviour, enhanced user interface, and means-end coherence. They deal with the problem of how to get from some current state to a desired state through a sequence of actions. The way planning systems work is studied extensively in the literature, and can be described as the process of searching for an appropriate plan using a specification of the available actions and their preconditions and effects.

One of the most well-known planning systems in use is *STRIPS*, originally presented by Fikes and Nilsson [1971]. It is described as:

> "*(...)STRIPS belongs to the class of problem solvers that search a space of 'world models' to find one in which a given goal is achieved. For any world model, we assume there exists a set of applicable operators each of which transforms the world model to some other world model. The task of the problem solver is to find some composition of operators that transforms a given initial world model in one that satisfies some particular goal condition.*"

In addition, there are several future extensions to STRIPS in the literature, grouped in the field of *automated planning and scheduling*. Ghallab et al. [2004] provides an extensive view on the topic of automated planning, introduces several implementation approaches and presents evolutions and alternatives to STRIP-like planning systems.

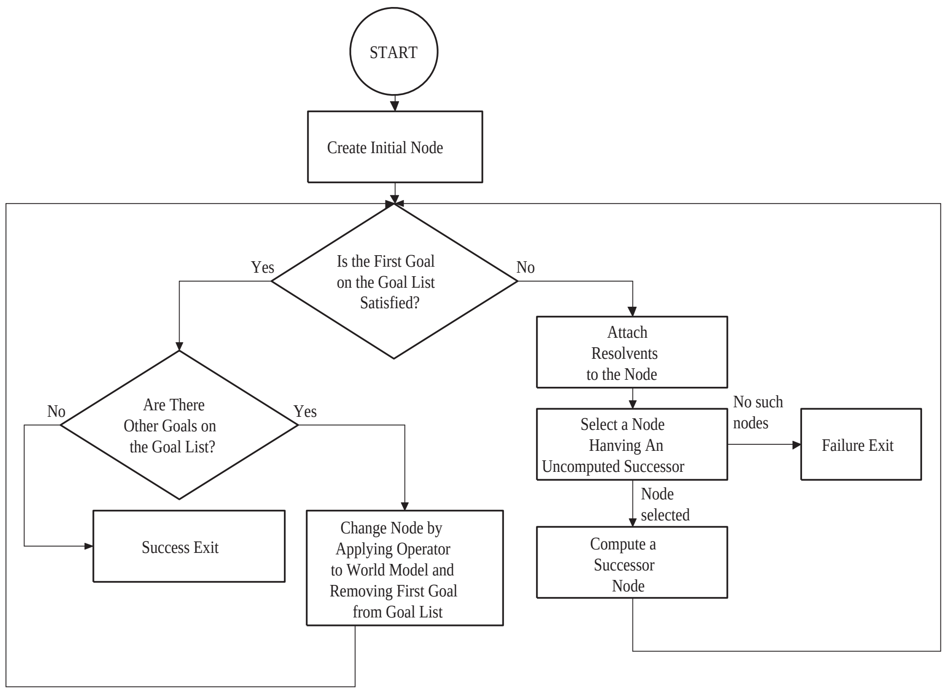

START
Create Initial Node
Is the First Goal on the Goal List Satisfied?
Yes
No
Are There Other Goals on the Goal List?
No
Yes
Success Exit
Change Node by Applying Operator to World Model and Removing First Goal from Goal List
Attach Resolvents to the Node
Select a Node Hanving An Uncomputed Successor
No such nodes
Failure Exit
Node selected
Compute a Successor Node

Figure 2.2: Diagram of STRIPS Executive (from [Nilsson and Fikes, 1970])

In summary, a typical STRIPS-like planner takes three inputs: a description of the initial state of the world, a description of the desired goal, and a set of possible actions. The planner produces a sequence of actions that lead from the initial state to a state meeting the goal. An alternative language for describing planning problems is that of hierarchical task networks, in which a set of tasks is given, and each task can be either realised by a primitive action or decomposed in a set of other tasks.

Analysis and Shortcomings

Figure 2.2 depicts the STRIPS' executive module. It provides the basis for several STRIPS-like planners that came after its developments. It works by resolving plans to achieve desired goals, however does not implement any structure to provide reactiveness to environmental changes. So once it adopts a goal and start to plan for it, it will continue to execute that line of processing regardless of the situation evolving.

Moreover, the process of looking for ideal solutions in large search spaces can become unfeasible with regards to time and computational resources. The investigation of efficient ways to implement this search and the development of enhanced heuristics to improve performance is the concern of several works in the field. Some later developments unify the idea of reactive planning – i.e.

planning in reaction to conditions of the environment – and pro-active planning – i.e. goal oriented planning. That is the case of *Procedural Reasoning System* (PRS), which I introduce in the next sub-section.

Finally, the planning process must avoid the pitfalls of overcommitted planning systems that sacrifice reactiveness to reach a perfect plan. Hence, it has to be reactive to the changing environment conditions. In this work, I propose an extended BDI-model inference system equipped with a *planning module* that makes the connection between the reactive and pro-active behaviours of the application's deliberation. In principle, it is activated by events triggered by conditions that request either the deliberation or reconsideration of deliberative attitudes.

2.4.3 Practical Reasoning

The problem of scalability in planning systems is mainly due to the model finding multiple options to reach the same goal. With the practical reasoning model the application focuses on a single line of reasoning; the problem, of course, is how to select which line of reasoning to follow. For that, [Bratman, 1987, page 35] says that:

> "*(...) a central problem for a theory of intention is to provide a satisfactory model of the relation between two kinds of practical reasoning: the weighing of desire-belief reason for and against various options, and reasoning from a prior intention to intentions concerning means, preliminary steps, or more specific courses of actions. (...) Practical reasoning, then, has two levels: prior intentions and plans pose problems and provide a filter on options that are potential solutions to those problems; desire-belief reasons enter as considerations to be weighted in deliberating between relevant and admissible options.*"

This view leads to the concept of "on-demand plan selection" based on cost of actions. That is, the inference system selects the line of reasoning to follow based on desire-belief reasons and the selection of the planning rule with lower costs. Of course, it requires the representation of the costs and a method to estimate the cost of a line of reasoning prior to adopting that path. In practical implementations, such as the one proposed in this work, it means that it is not possible to assure the *best cost*; thus, it is possible that incomplete information the planner system ends up taking a path that has the immediate lower cost, but adds up to a higher cost as the situation progresses.

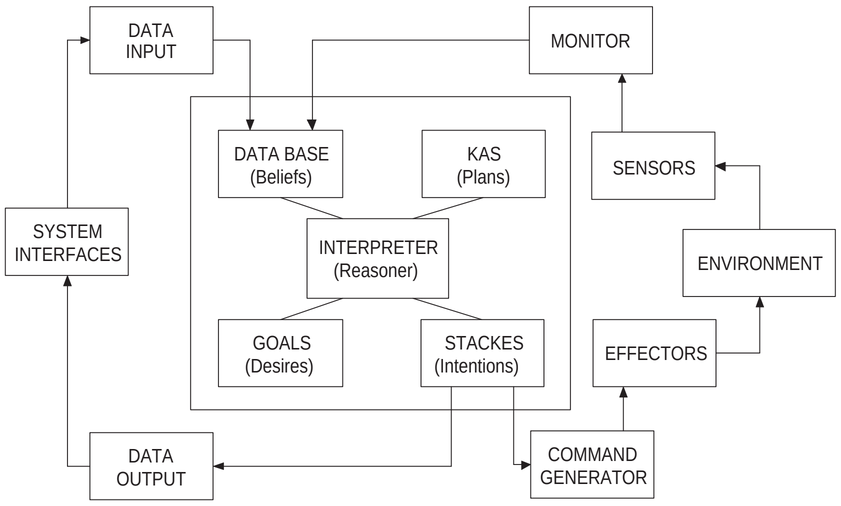


Figure 2.3: PRS System Architecture (from [Georgeff, 1987])

In other words, this plan selection process results in minimum locals derived from sub-goals and conditions.

The practical reasoning model introduces advantages in relation to *plan ahead* planning system, as described in Rao [1996] and summarised below:

1. *promotes the balance between reactiveness and pro-activeness* by allowing the inference system to evaluate the environmental conditions at each planning step;

2. *allows adaptiveness* of the planning process by reconsidering the current planning threads in between each planning step;

3. *it is more scalable*, because only one planning path will be elaborated at each step; so it avoids the issues in plan ahead planning systems that require a long time and large resources to compute the plan tree.

These advantages are aligned with the requirements to implement intelligent mobile services, therefore suggest integrating this model with concepts from context-awareness and cognitive behaviour, and applying the resulting model towards developing such applications.

Analysis and Shortcomings

Procedural Reasoning System (PRS), introduced in Georgeff [1987], was one of the first attempts to implement an application based on the practical reasoning model. Figure 2.3 depicts the basic structures in this system, which is explained in the following quote.

> *"The system consists of a data base containing current beliefs or facts about the world, a set of current goals or desires to be realised, a set of procedures (which, for historical reasons, are called knowledge areas or KAs) describing how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations, and an interpreter (or inference mechanism) for manipulating these components. At any moment, the system will also have a process stack (containing all currently active KAs) which can be viewed as the system's current intentions for achieving its goals or reacting to some observed situation."*

The PRS interpreter runs the entire system in a relatively simple way:

1. at any particular time, certain goals are active in the system and certain beliefs are held in the system database;

2. given these current goals and beliefs, a subset of plans are applicable; one of these plans is chosen and placed on the process stack for execution;

3. while executing this plan, new subgoals will be posted and new beliefs derived; when new goals are pushed onto the goal stack, the interpreter checks to see if any of the new plans are relevant; if so, it chooses one, places it on the process stack, and starts its execution;

4. whenever a new belief is added to the database, the interpreter will perform appropriate consistency maintenance procedures, possibly activating other relevant plans. During this process, various meta-level plans may also be called upon to make choices among alternative paths of execution, choose among multiple applicable plans, decompose composite goals into achievable components, and make other decisions.

The process interleaves plan selection, formation, and execution. In short, the system forms a partial overall plan, determines a means of accomplishing the first subgoal of the plan, acts on this, further expands the near-term plan of action, executes further, and so on. PRS will try to fulfil any intentions it has previously decided upon unless some new fact or request activates a new plan. But if some important new fact or request does become known, PRS will reassess its goals and intentions and perhaps choose to work on something else.

The PRS system is the forefather of current BDI-model architectures widely studied in the agent community. In fact, they share a number of common features. In this work, I am proposing an extended BDI-model that shares several similarities with the PRS-model in the way they process changes in the environment and reconsider the current deliberations.

The support provided by PRS to the requirements to implement intelligent mobile services is summarised in Table 2.3.

2.4.4 Belief-Desire-Intention Model

The BDI-based model is a specialisation of agents with practical reasoning deliberation cycles and a specific format for knowledge representation. It encompasses Michael Bratman's respectable philosophical model of human practical reasoning, originally developed in Bratman [1987]. The model represents the knowledge structures in the following way:

- *beliefs* represent the state of the environment, such as environmental variables (e.g. current location, time), user's profile (e.g. agenda, preferences) and activities;
- *desires* describe the outlines for the application behaviour, that is the generic goals that the application should pursue when proper conditions arrive;
- *intentions* represent the goals to which the application is committed to; for example, if it is 9:50am and there is a meeting in room A110 about project A at 10am, and the user has a generic desire to have documents about any project accessible when he walks into a project-related meeting; then the application deliberates the intention to download the documents related to project A for the meeting at 10am;
- *plans* represent the practical reasoning rules on *how* agents infer *what to do* to achieve a goal; for example, in the intention above the agent must formulate the plan to download the documents; a plan contains a set of sub-plans (or sub-goals) that would lead to the main goal.

Figure 2.4(A) depicts the architecture of a BDI-model agent. The deliberation cycle sits in between the knowledge representation structures and animates the agent. It works based on the concept of a "check-deliberate-action" loop, previously described in Rao and Georgeff [1995], and depicted in Figure 2.4(B).

Rao and Georgeff [1995] suggests that an important feature of BDI-model deliberation cycle is its commitment to the current deliberation and, consequently, its inflexibility to adapt. That work mentions that:

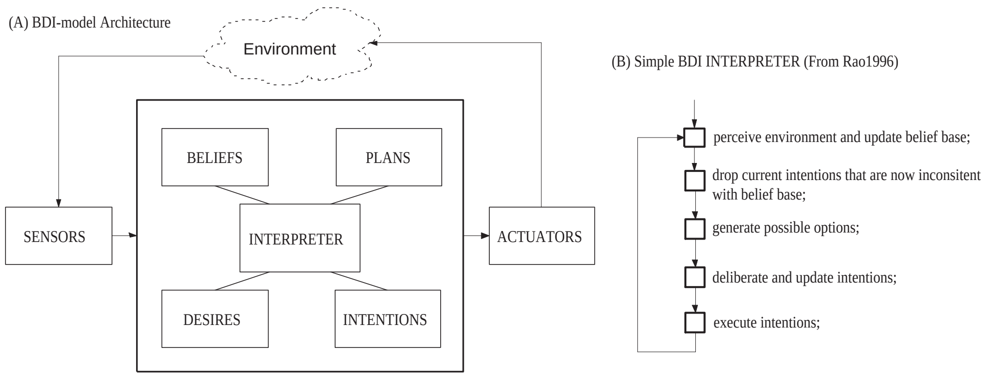

(A) BDI-model Architecture
Environment
BELIEFS
PLANS
SENSORS
INTERPRETER
ACTUATORS
DESIRES
INTENTIONS
(B) Simple BDI INTERPRETER (From Rao1996)
perceive environment and update belief base;
drop current intentions that are now inconsitent with belief base;
generate possible options;
deliberate and update intentions;
execute intentions;

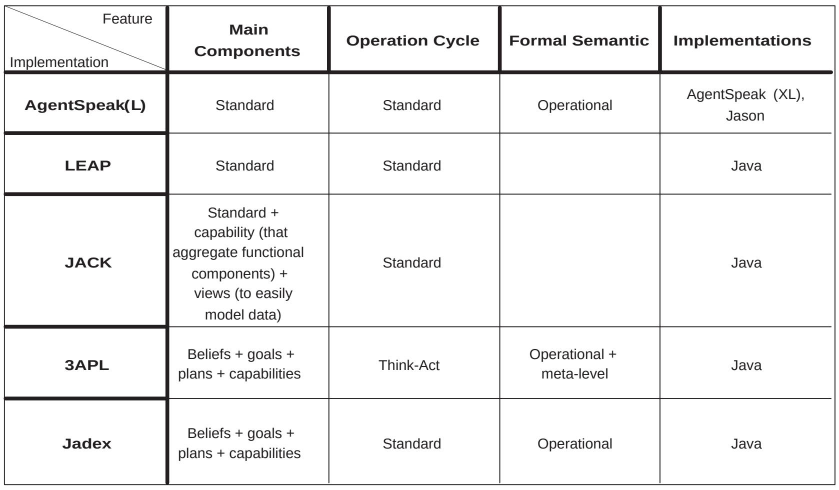
Figure 2.4: BDI System Architecture and Deliberation Cycle (from Rao and Georgeff [1995])

> "*(...) an important aspect of a BDI architecture is the notion of commitment to previous decisions. A commitment embodies the balance between the reactivity and goal-directedness of an agent-oriented system. In a continuously changing environment, commitment lends a certain sense of stability to the reasoning process of an agent. This results in savings in computational effort and hence better overall performance. A commitment usually has two parts to it: one is the condition that the agent is committed to maintain, called* the commitment condition*, and the second is the condition under which the agent gives up the commitment, called* the termination condition*. As the agent has no direct control over its beliefs and desires, there is no way that it can adopt or effectively realise a commitment strategy over these attitudes. However, an agent can choose what to do with its intentions.*"

The BDI-model provides the advantages of a practical reasoning system plus an improved knowledge representation structure. I suggest integrating this model with concepts from context-awareness and cognitive behaviour, and applying the resulting model to developing intelligent mobile services. The support the BDI-model provides for the requirements to implement such services is summarised in Table 2.3.

Analysis and Shortcomings

Mascardi et al. [2005] presents a survey of BDI-style agent platforms in the literature. I analyse the features provided by the most popular implementations in Table 2.5, based on the following criteria:



Feature / Implementation	Main Components	Operation Cycle	Formal Semantic	Implementations
AgentSpeak(L)	Standard	Standard	Operational	AgentSpeak (XL), Jason
LEAP	Standard	Standard		Java
JACK	Standard + capability (that aggregate functional components) + views (to easily model data)	Standard		Java
3APL	Beliefs + goals + plans + capabilities	Think-Act	Operational + meta-level	Java
Jadex	Beliefs + goals + plans + capabilities	Standard	Operational	Java

Figure 2.5: Analysis of BDI-Model Implementations

- *main components*: relates the components in the agent structure, that is beliefs, desires, intentions, and plan rules or plan receipts as they are called in some works;

- *operation cycle*: describes the format of the deliberation cycle where *standard* refers to the ones that follow (somehow) the "check-deliberate-action" from Rao and Georgeff [1995]; and *think-act* are based on a two-part cycle corresponding to a first-phase of practical reasoning using practical reasoning rules, and a second execution phase in which the agent performs some action;

- *formal semantics*: describes where a semantic logic is provided with the language, with *operational* meaning that the semantics for operation controls (e.g. transitional logic) is provided, and; *meta-level* meaning the semantics for meta -operation control are also provided;

- *implementations*: describes current implementations.

AgentSpeak(L) is an agent architecture and language with explicit representations of beliefs, goals, and intentions, which was initially introduced by Rao and Georgeff [1995], who provided a language for specifying BDI agents with a sketch of an abstract interpreter for programs written in the language. It provided the first bridge to the gap between BDI theory and practice. The implementation has clear, precise logical semantics and is described in a computationally-viable way, with AgentSpeak(L) ports to Java

2 Micro Edition[5], such as in *AbiMA Project* [Rahwan et al., 2003]. Popular implementations are: AgentSpeak(XL), introduced in [Bordini et al., 2002], and; *Jason*, introduced in [Bordini et al., 2007].

Lightweight Extensible Agent Platform (LEAP), introduced in Bergenti et al. [2001], is the first attempt to implement a FIPA agent platform that runs seamlessly on both mobile and fixed devices, over both wireless and wired networks. Having many strengths, this platform satisfies the requirements for intelligent support, collaboration and personal assistance. Although the platform can be adapted to integrate to context-awareness and device interface, this feature is not clearly defined in the product.

JACK, introduced in [Padgham and Winikoff, 2002], is a commercial agent development platform developed by *Agent Oriented Software*[6]. Including an agent-oriented programming language that is a superset of Java, JACK uses a modular approach to build agents. Agents communicate with each other using a portal. All communication is achieved in the form of events, which are handled by plans; the body of a plan contains regular Java code, augmented with JACK-specific reasoning methods, which are incorporated into Java code at compile time.

3APL, and its lightweight implementation 3APL-M, introduced in Hindriks [2001] and [Koch and Rahwan, 2005] respectively, provide programming constructs for implementing agents' beliefs, goals, and capabilities. It uses practical reasoning rules to generate plans for achieving agents' goals and for updating or revising these plans. An *interpreter* that deliberates on the cognitive attitudes of that agent executes the program. 3APL-M is implemented using Java 2 Micro Edition.

Jadex, presented in [Braubach et al., 2004], is a "*Jadex is a Java based, FIPA compliant agent environment, and allows to develop goal oriented agents following the BDI model. Jadex provides a framework and a set of development tools to simplify the creation and testing of agents*[7]." *Jadex* is based on the *JADE Agent Framework*, an open source development. *JADE* provides the platform architecture and the core services and message transport mechanisms required by the *FIPA* specifications. Further advantages of using JADE arise from its stability, features such as agent deployment and debugging tools, and its large and active user base.

[5] *Java 2 Micro Edition (J2ME)* is a Java platform runtime environment widely adopted by mobile computing hardware manufactures; see web-site at http://java.sun.com/j2me.

[6] *Agent Oriented Software*: web-site at http://www.agent-software.com, accessed in Nov-2008.

[7] *Jadex*: web-site at http://vsis-www.informatik.uni-hamburg.de/projects/jadex, accessed Nov-2008.

These platforms share a number of common features, such as support to BDI-model elements' representation – i.e. beliefs, desires, plans –, an interpreter to execute the program, and an interface to collect data and to support user interface.

Shortcomings. The BDI-model provides the tools to implement the reconsideration conditions in between the planning steps. However, the model does not specify *when* to reconsider the existing deliberations. That is, in order to be able to determine at what point the deliberation process should reconsider its current deliberations, the architecture must provide a module that observers the environmental conditions and decides whether it is time to reconsider the deliberation. The absence of the *context observer* module implies that the application is unable to react to impromptu environmental changes.

In this work, I propose to extend the BDI-model with an inference system equipped to take advantage of expectable context situations, where *expectancy* relates to the ability to estimate proximity to given situations. This directly relates to the concept of window of opportunity, reducing the problem to the issue of how to determine when the user is in or out that area. In the case of mobile services, it is possible to calculate proximity based on location and time. However, I acknowledge that this restriction leads to solutions that operate in the dimension of time and space, which is enough for mobile services but insufficient to provide a generic solution. In chapter 5, after having presented the overall solution, I discuss how to extend this functionality to encompass other dimensions of context information.

2.5 Conclusion

I summarise my line of thought below:

- *context-awareness* is an indispensable feature to create MPAs capable to react to environmental changes;
- it is possible to implement context-aware applications based on *sensor-interpreter-application* models;
- the *interpreters* are the key element in this model towards "smart (context-aware) applications", being responsible for mapping between context representations and for performing complex inferences on collections of context;

- *agent-based computing* provides several inherent solutions, such as support for knowledge representations, situatedness, enhanced deliberation, and proactiveness;

- *the BDI-model* provides structures that support the development of MPAs;

- there are works on how to *integrate context-awareness and BDI-model*, making this the engine for the *sophisticated interpreter* in a context-awareness centric view.

Opportunities exist to combine solutions from context-awareness and agent-based computing for the development of intelligent mobile services; resulting in a framework where agent-based applications are able to deliver proactive, self-tuning, context-aware, autonomous, personal assistance to the mobile user.

The BDI-model provides the features to support the development of mobile services. This model provides the tools to implement the reconsideration conditions in between the planning steps, however, the model does not specify *when* to reconsider the existing deliberations. Moreover, current implementations lack the level of reactiveness required to support highly dynamic environments. Although the initial designs were inspired by reactive approaches, the focus on BDI-model evolution was towards pro-active systems equipped to provide adaptive behaviour in detriment to reactiveness. The current designs work based on deliberation cycles that implement a "check-deliberate-action" loop, which implies in applications overcommitted to the planning phase.

As a possible solution, I propose to optimise the design by equipping the inference system with the structures to allow the impromptu reaction to environmental changes. In addition, the deliberation cycle must be able to reconsider its intentions and plans in real-time, in reaction to these events. I detail this proposal in the next chapter.

Chapter 3

Conceptual Model

"I can't work without a model. I won't say I turn my back on nature ruthlessly in order to turn a study into a picture, arranging the colours, enlarging and simplifying; but in the matter of form I am too afraid of departing from the possible and the true."
Vincent van Gogh.

3.1 Introduction

In the previous chapter, I argued that current BDI-modelled agent implementations do not inherently support mobile services. However, this technology does provide desirable features that are strongly aligned to the requirements of the target applications. I proposed the following extension in order to support this development: (i) enhanced methods for impromptu reconsideration of in-processing elements; (ii) structures to take advantage of events from window of opportunity; (iii) controls to maintain application coherence amidst environmental instabilities, and; (iv) computing optimisation to allow these applications to execute on constraint resources. These requirements are linked to one key issue: how to balance reactiveness and goal-orientated behaviour in the inference system.

In this chapter, I continue that discussion and further detail the related concepts. My aim is to provide a clear picture on *why* agent applications must be able to adapt their course of actions; *what actions* are implied by adaptation, and; *what structures* are required to support these actions. I also introduce a syntax for describing the elements and operations.

The chapter is organised as follows. The next section presents a conceptual problem scenario and elaborates on related issues. Section 3.3 introduces the language to describe the elements, while Section 3.4 describes the operations.

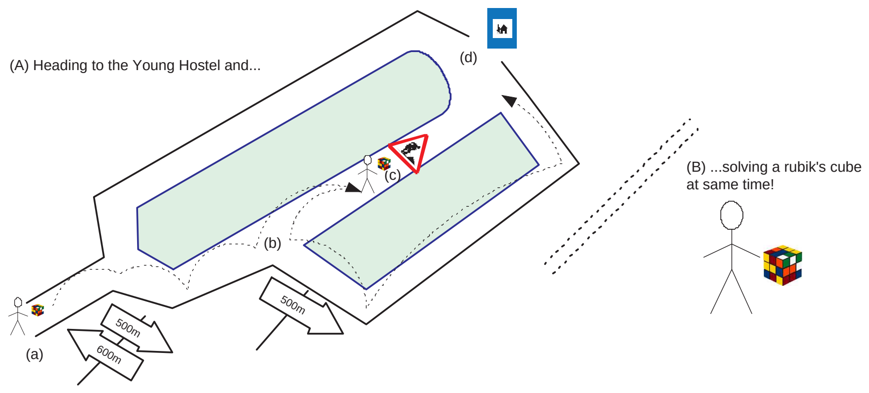

(A) Heading to the Young Hostel and...
(d)
(c)
(b)
(a)
500m
600m
500m
(B) ...solving a rubik's cube at same time!

Figure 3.1: Conceptual Problem Scenario

The conceptual model is outlined in Section 3.5 before the chapter's concludes with Section 3.6.

3.2 Concepts

In this section, I discuss the key concepts of goal-oriented behaviour, unexpected events, concurrent intentions, planning, conflicts, prioritisation, reconsideration, adaptation, and others. For that, let us consider the illustrative example depicted in Figure 3.1 and described below:

Example 2. (tourist walking in a park)

> *A tourist is walking through a park, heading to his hostel. He is also trying to solve a "Rubik's Cube" while he walks*[1]*. At a certain point (a), the path branches. A sign indicates that the hostel is 600 metres via the left track (i.e. (a)-left) , and 500 metres via the right track (i.e. (a)-right). Unaware of any other information, the tourist decides to take the shorter track, that is (a)-right.*
>
> *After walking 100 metres, there is another bifurcation at point (b). A new sign indicates that there is another 500 metres to the hostel by the right-hand track. Based on the original sign, the tourist concludes that the hostel should only be 400 metres via the current*

[1] *Rubik's Cube*: Korf [1997] describes how to solve this game in up to 26 moves!

> *track. He walks another 200 metres along (b)-left then hits a road block, which was unexpected.*
>
> *At this point the tourist has to decide on what to do next. He keeps his desire to reach the hostel, of course, but he must review his current plan. He puts aside his Rubik's Cube and focuses on searching for alternatives. He finds two possibilities: (i) to go back to point (b) and take the alternate path (b)-right, or (ii) to go back to point (a) and take the original alternate path, (a)-left.*
>
> *There is no guarantee that either of these tracks is not also a dead-end. Therefore, the decision is based on distance, considering the information from the road signs. He decides to go back to point (b) and take (b)-right, which is the shortest path from his current point. Once he has decided what to do, the tourist resumes his Rubik's Cube game and keeps walking.*

This scenario illustrates several issues related to reactiveness and goal-oriented behaviour. It highlights elements of Belief-Desire-Intention (BDI) and deliberation, such as means-end coherence. In addition, it can be seen as a metaphor for a planning system and its issues.

The *Belief-Desire-Intention model*, proposed in Bratman [1987], is a natural approach to represent the elements of this scenario. In this representation, *beliefs* describe the state of the environment; *desires* define the application's long-term goals, and; *intentions* are the goals the agent is committed to achieve. The tourist's acts are directed to goals – i.e. reach the hostel and resolve the Rubik's Cube – so it can be described as *goal-oriented behaviour.* In addition, the tourist's deliberation is running *concurrent intentions.* Sometimes, in reaction to *unexpected events* when executing these intentions (e.g. the road block) the tourist has to resolve *conflicts/* In this case, i.e. the inability to resolve the Rubik's Cube and run a map search at same time. In face of resource conflicts, he *prioritises* the most important tasks and *reconsiders* his current intentions.

The tourist has to decide which committed goals to follow. Next, he formulates the sequence of actions that compose the plan. This process is named *plan formation.* However, he can only plan for a few steps ahead. In the example, the committed goal is to walk in the hostel's direction, following the road signs. He formulates the following plan: walk, follow the signs, turn, keep walking. Nevertheless, the existing information is insufficient to form a complete plan. That is, he knows his next actions as far as *walk and follow the signs.* At each sign he must decide upon his next sub-goals – i.e. turn or keep walking – according to the environmental information. This planning behaviour is described as *plan on-demand*; this is a predominant feature in current BDI-modelled agent implementations.

3.2.1 Belief-Desire-Intention Model

Firstly, to support my presentation, I recall the concepts of BDI elements - beliefs, desires, intentions, and plans - and introduce informal descriptions. I infer the following elements from the scenario:

- *beliefs*: represent the state of the environment; the set of beliefs constitutes the model of the world; it contains what the tourist perceives about the surrounding situation and everything else he knows about the world, such as current location, time, costs of actions, elements of the environment, personal history, and information about relatives[2]; however, only a sub-set of this information is important for the current deliberations.

- *desires*: define the application's long-term goals, which are activated when proper conditions are met; note that it is possible to have conflicting desires dormant in the desire base, which cannot be activated concurrently; it is not feasible for an agent to continually evaluate the conditions for all of its desired goals, thus, at some point, the agent sits on achieving certain courses of action; these are the committed goals, or *intentions.*

- *goals*: are the goals the agent is committed to achieve. They provide a characteristic of stability, in the sense that an agent will not reconsider previously formed intentions. If an agent wants to adopt new intentions, they should not conflict with existing ones; for example, the tourist forms the intention to go to the hostel, which is not conflicting with his intention to resolve the Rubik's Cube; however, at a certain point, the latter conflicts with the new intention of searching for a solution on the map. In this situation, it will have to *prioritise* the most important one; these activities are part of what Bratman [1987] calls *intention reconsideration.*

- *intentions*: describe the set of actions to be taken in order to achieve a goal; an agent concludes that it is reasonable to adopt a plan set of actions - as long as he believes the plan will achieve a particular situation and its adoption condition (the *guard*) holds true with the

[2]*Elements of context information*: Graham and Kjeldskov [2003] describes eight dimensions of context information to be considered in mobile setups: time, absolute location, relative location, objects present, activity, social setting, environment, and culture.

model of the world; [Bratman, 1987, page 31] suggests that "*[plans must be] strongly consistent relative to the beliefs*"; meaning that once the model of the world changes, plans might have to be *revised*. In the illustrative scenario, that happens when the path turns into a dead-end.

Next, I describe several concepts related to practical reasoning and adaptation.

Goal-oriented Behaviour

From the problem scenario, one can intuitively conclude that the tourist is deliberating towards his desired goals: (A) get to a safe place to rest, and (B) resolve the mind-challenging Rubik's Cube. Cohen and Levesque [1990] describes a *goal* as: "*a state the agent would like (most) to be in if it is not there yet, or to maintain in case it is in that state*".

The agent's purposeful behaviour, or commitment, provides a level of stability to the reasoning process. That is, the tourist knows he needs to do (A) and (B), which keeps him from switching between these and other unimportant goals. In other words, intending to do p implies a commitment of having p done. Therefore, intentions are future-directed; or a commitment to success, as described in that work. When deliberating on intentions, the agent analyses the required resources to implement the actions, deliberates on the sequence of actions to follow, and executes the basic actions upon the environment.

Concurrent Intentions

An agent can process more than one intention at the same time, as long as they are not conflicting. Our capacity to plan for several intentions simultaneously seems trivial, as we separate the processing spaces. For example, the tourist's intention (A) get to a safe place to rest, and; (B) to solve the Rubik's Cube. In fact, these goals exist in different domains. That is, while (A) requires the use of physical resources, (B) predominantly requires mental resources.

However, at a certain point there is a new intention (C) for map searching. The agent *pauses* the deliberation for (B) and switches to (C). The rationale is intuitive: both (B) and (C) draw on mental resources. Although the agent does desire to fulfil both (B) and (C), their simultaneous execution is counter-productive. In other words, one cannot study a map while resolving a Rubik's Cube. Cohen and Levesque [1990] states the following on this topic:

"(...) *we do not include an operator for wanting, since desires need not to be consistent. Although desires certainly play an important* *role in determining goals and intentions, we assume that once an agent has sorted out his possibly inconsistent desires in deciding what he wishes to achieve, the worlds he will be striving for are consistent*".

The decision to switch between goals happens based on *conflicts* and *prioritisation.* In the tourist scenario, goal (C) has higher priority than goal (B), however the prioritisation process is also context-dependent. Thus, once the agent thinks he is back on the right path, then goal (B)'s priority becomes higher than (C)'s . That is, if the agent is already walking the right path, then why keep looking at the map?

The possibility of switching between goals – i.e. to pause (B), start (C), then pause (C), and re-start (B) – is what I call *deliberation checkpointing.* This behaviour is coherent only if there are no visible conflicts. However, it is possible that there are *unknown conflicts*, such as situations the agent has not yet learned from experience. In this case, the execution of parallel, conflicting goals renders *exceptions* (to be defined). This situation presents an opportunity to learn and adjust the deliberative behaviour; a feature that is part of *adaptation behaviour*, introduced later in this chapter.

It is intuitive that *goal switching* implies *processing overhead.* When referring to human behaviour, this situation is often called as *distraction.* For instance, it happens when we receive a phone call in the middle of a meeting. We can easily pause our current conversation, pick up the phone, have the side-line conversation, and come back to the meeting at the same point. However, it is quite common to have a delay until we are finally able to re-take the original line of thought. The impact of the processing overhead must be considered in the computation system to measure the efficiency of the implementation.

Unexpected Events

Unexpected events are a feature of dynamic environments. In fact, the more dynamic the environment, the more likely that something will happen unexpectedly. Mobile environments highlight this issue as open environments and mobility tends to increase the amount of information that one perceives. Other factors to consider include the number of sensors, and the degree of dynamism. One distinguishing feature in agents is *situatedness*, that is the capacity of perceiving and representing the environment around it. Software agents are applications designed to operate i.e. able to *maintain coherence*- in an unstable, unpredictable environment.

Obviously, the *unexpected* is inversely related to "*things that one expects*", that is, the probable[3]. In the illustrative example, at the first bifurcation the probabilities of (A) get to a safe place to rest using either track were unknown; at that point it is a guess, whichever course of action is taken. After the tourist has learned about the dead-end, the knowledge about the probabilities of achieving the hostel are adjusted accordingly, and the dead-end would no longer be unexpected if that path were chosen again.

In other words, one can say that unexpected situations happen when the agent lacks information about the environment. This situation exists because either the information cannot be modelled beforehand, or it is missing. Another possibility is that the environment has changed, thus the pre-modelled information is no longer consistent with the current environment.

The agent must be prepared to handle the unexpected in order to maintain coherence and stability in the face of the improbable!

Conflicts

Conflicts exist when resources cannot be shared between two plans, either within the same agent or between different agents! For example, the tourist's intellectual power cannot be shared between the activities of resolving the Rubik's Cube and running a map search.

Conflicts presuppose simultaneous actions, although the inverse is not necessarily true. Conflicts are a by-product of unmet needs and unrecognised differences, and result from perceived incompatible plans, goals or actions. These situations are inherent to dynamic environments, where events and understandings constantly re-structure and re-interpret the past, present and future. At a rational level, all deliberation conflicts can be described as having origins, dynamics, processes and outcomes; thus, the events that led to the conflicting situation can be learnt and behaviour adjusted. However, in practical reasoning, the agent is not questioning the source of the conflict, but what to do to solve it.

An agent can hold conflicting desires, however, his intentions should be conflict-free. This constraint imposes a *filter of admissibility* for further intentions the agent can adopt. For example, the tourist cannot have an intention to go to (d) then, suddenly, form an intention to go to another place and keep switching between these two intentions; he would enter an endless, counter-productive goal-switching loop. In addition, there are the resource conflicts

[3]*Probability versus Decision Theory*: several works on decision theory resort to *probability theory* to subsidise the decision process. In this work, I am not exploiting probability as an element of the deliberation behaviour, for practical reasons. Different types of deliberation constructs, such as probabilistic graphical model, would have to be employed to explore this feature.

between basic actions of two simultaneous intentions. For example, it is not productive to switch looking between the map and the cube, i.e. giving resources to a lesser priority task whilst a higher priority is unfilled. In this work, I classify two groups of conflicts, as described below:

- *goal conflicts* describe the situation where the agent cannot pursue two goals at same time. For example, it is intuitive that the agent should not try to move to two different bases at the same time, otherwise it will remain in a loop: the goal to $go(a)$ is conflicting with the goal to $go(b)$. The actions to $moveNorth()$ and $moveSouth()$ are not *physically* conflicting as long as they are interleaved – that is, one can move north and then move south, one action after the other. That is not conflicting, just counter-productive. The agent must know that two goals are conflicting – from experience and reinforcement learning – and prioritise the more important one.

- *resource conflicts* describe the conflicts between basic actions. If the agent is resolving the action that leads to a goal, and this action is conflicting with one associated with another committed goal, they cannot be executed simultaneously. For example, the traveller cannot resolve the Rubik's Cube and look at the map at the same time, because these two activities share the same resource (i.e. his mental attention).

Prioritisation

Prioritisation is one of the possible solutions for conflict resolution. Converting desires into intentions requires verifying the possible conflicts that these newly conceived intentions can cause to existing ones. This problem is intuitively obvious: sometimes the user might have numerous goals that cannot be achieved simultaneously; in which case, the agent makes the decision on how to *prioritise goals.*

For example, if the tourist is unable to prioritise, he would keep deliberating on conflicting intentions, such as (B) to solve the Rubik's game, and; (C) to find a solution using the map. It seems obvious that trying to resolve (B) and (C) at the same time is counter-productive; the likely outcome being that the agent would not solve the cube anytime soon and would take longer to reach the destination. This issue relates to the problem of distraction and the processing overhead caused by goal switching, described above.

A better approach is to pause (B), focus on (C) and, once (C) is completed, resume (B). To take this decision the agent must know that (C) has higher priority than (B). It is assumed that, based on past experience, generalisations, common sense or any other means, the agent has a representation of priority between goals as part of his knowledge. Using this knowledge it decides which intention takes precedence in a given situation, and what to do with the other intention.

Intention Reconsideration

Intention reconsideration is an outcome of the prioritisation process, and part of the conflict resolution. The agent has to decide what to do with the intention that has been demoted during the conflict resolution. [Bratman, 1987, page 62] suggests that:

> "*(...)To reconsider a prior intention to A is not just to entertain the possibility of a change in the intention. It is seriously to re-open the question whether to A, so that this is now a matter that needs to be settled anew. One withdraws the intention from the background against which one deliberates. One no longer appeals to that intention in one's practical reasoning.*"

In the problem scenario, the tourist reconsiders the intention to resolve the Rubik's Cube when it conflicts with the map search activity. However, he does not resolve to throw the cube away. Nor does he change his behaviour on how to resolve the cube. The current plan to resolve the Rubik's Cube is well formed and consistent, however, changes in the situation led to its demotion. In this case, it is sufficient to pause the planning process for that intention, keeping the current mind status, allowing the intention to be resumed later when the conditions are conducive.

Plan Revision

Plan revision is another aspect of intention reconsideration. It happens when the plan is either ill-formed or its execution has failed.

The rationale is as follows: agents are situated in some environment which can change during the execution of the agent, requiring flexibility of the problem solving behaviour, i.e. the agent should be able to respond adequately to changes in its environment, however, programming flexible behaviour is not a trivial task. For example, standard procedural languages assume that the environment does not change while some procedure is executing. If problems occur during the execution, then the program might throw an exception and terminate. While this solution might work well for many applications, it fails to support *autonomy* in agents' solutions, which should maintain the application's coherence.

For example, in the illustrative scenario, the planning and execution of intention (A) walk to the hostel fails when the tourist hits a road block. Although the goal itself is of high priority for the tourist, the plan is ill-formed in the sense that it did not consider the road block (due to the lack of information).

To resolve these situations, agents have *plan repair rules* as part of their knowledge, which are used to decide how to recover from exceptional situations, looking for alternatives that maximise resources saving and the chance for successful outputs. Nebel and Koehler [1995] argued that "*modifying an existing plan is (worst-case) no more efficient than a complete replanning in theory; (...); but the idea is that in practice plan repair can be more efficient [than drop the plan and run a complete replanning]*". This is the case in the illustrative scenario. For the tourist to replan on *how to get from (a) to (d)* is, in practice, less efficient than the alternative of backtracking to the last decision, at bifurcation (b), and taking the alternative path.

van Riemsdijk [2006] provides a detailed discussion on this topic and mentions that:

> "*Goals and plan revision form two important aspects of the presented cognitive programming languages. These aspects (...) can be linked by viewing them as both facilitating the programming of flexible agents. Agents need to have a certain level of flexibility, since they are often expected to operate in dynamic environments.*"

I detail the issues related to plan revision when presenting the operations of the planning process in Section 3.4.

3.2.2 Planning

Planning is the process of finding solutions to achieve goals. In agent research terminology, it means that the agent finds the set of possible plans from his set of plan rules and decides on which one to apply. There are different approaches to select which plan to follow. A common solution is to maximise utility; to select the one that is less costly. Doyle and Thomason [1999] says that "‘*(...) good decisions are formally characterised as actions that maximise expected utility, a notion involving both belief and goodness.*" This approach is feasible if the agent knows the sequence of basic actions to be taken, and the costs associated with executing these actions.

For instance, in the illustrative scenario the tourist is informed by the road signs about the distances (i.e. cost) of each road. He has no information about the probabilities of reaching the destination using a certain track, however.

Therefore, his decision is based on minimising action costs, which means taking the shortest paths. Clearly, the final solution is not the optimal, as the agent could have taken path (a)-left from the start and walked 600 metres rather than walking a total of 1000 m (backtracking along (b)-left and taking (b)-right from point (b), assuming (b)-right actually arrives at (d)). However, the tourist did not have the complete information about the tracks so he could not weigh probabilities in his decision. Although, the implemented solution ultimately lead to poor results, the planning process behaviour is acceptable for the reasons explained below.

The classical decision theory described in Jeffrey [1990] assumes that the decision making agent is able to weigh all possible alternative courses of actions before choosing one of them. It is related to what I call *plan-ahead behaviour* in the next sub-section. The actions are decided based on two attributes: first, the probability of reaching a state by performing an action, and; second, the utility function that indicates the utility of states. A decision rule is used to determine which action to take. This approach has a clear limitation, however: it is useful only in closed system where the agent has complete information about costs and probabilities. This is not the case in open or complex systems, where bounded agents have restricted information of what lays ahead and have to choose and perform actions based on what is available. Moreover, in dynamic systems there is always the possibility that these attributes change over time; for instance, the road block could have been introduced after the tourist started to walk. Doyle and Thomason [1999] says that:

> "*The issues that arise in automated decision making recall some of the traditional philosophical questions but locate the debate in a new setting by focusing on the need for efficient reasoning in the very complex decision-making cases that can arise in human affairs. Quantitative representations of probability and utility, and procedures for computing with these representations, do provide an adequate framework for manual treatment of very simple decision problems, but they are less successful in more complicated and realistic cases.*"

Doyle and Thomason [1999] proposes an alternative approach, based on *qualitative decision theory*. It relaxes the assumption that both probabilities and utility functions should be present, and proposes a schema for reasoning with partial information. A rational agent is assumed to attempt to achieve the best possible world consistent with its (limited) knowledge. The theory proposes the state to be reached without indicating which actions to perform. The decision making agent is assumed to be a planning agent who can generate actions to reach the given state. The agent identifies alternatives, outcomes, probabilities, and utilities through an iterative process of "*hypothesising, testing, and refining a sequence of tentative formulations.*" This reflects the tourist's behaviour in deciding which track to take, which is a metaphor for the ability to formulate decisions in unforeseen settings.

Nonetheless, Dastani et al. [2003] proposes that:

> "*A criticism to both classical and qualitative decision theory is that they are inadequate for real time applications that decide actions in highly dynamic environments. In such applications, the decision making agent, which is assumed to be capable of performing a set of actions, should select and execute actions at each moment in time. As the highly dynamic environment changes either during the selection or the execution of actions, the decision making agent needs to deliberate to either reconsider the previous decisions, also called intentions, or continue the commitment to those decisions. (...) This approach has led to what is called BDI theory.*"

I recall that this work focuses on the extension of BDI-modelled agents, for the reasons introduced in Section 2.4.4. In what follows, I introduce the concept of *plan on-demand*, where the agent is able to balance reactiveness of planning with the information it has at a given moment to *proactiveness* of revising the sub-goals ahead in the search for a good plan.

Plan formation

The agent has to decide which plan to follow, and then forms the sequence of actions that compose the plan. The earlier scenario illustrates the issue: the tourist is committed to go to the hostel and his plan is to walk in its direction following the road signs. The sequence of actions is formed: walk, follow the signs, turn, keep walking. However, the information is insufficient to form a complete plan. He knows his next actions as far as *walk and follow the signs*, however, at each sign he must plan for the next sub-goals – i.e. turn or keep walking – according to the environmental information. If the tourist could count on complete information, for example from a map or mobile orientation system, then he could form the complete plan *a priori*. Hence, depending on the available information, there are two approaches to form the plans: *plan ahead*, which means to form the complete plan before the execution, and *plan on-demand*, which means to form the next (few) steps and start to execute, solving the next sub-goals as new information arrives. I discuss these approaches next.

Plan ahead behaviour. In this deliberative behaviour, the complete plan is formed before any execution. The multiple lines of thought are represented as a tree, as described by Emerson [1990]. Each node in the structure represents a certain state of the world, and each transition a primitive action made by the system, a primitive event occurring in the environment, or both. As the system has to act, it needs to select appropriate actions or procedures to execute from the various options available.

This behaviour is characterised as *completely proactive*; the planning system calculates all possible lines of resolutions – and problems, thereof – before taking any physical action. Of course, it presupposes complete information and static environment, as explained. Although the performance advantage is maximised, provided that a good planning structure and knowledge is available, this approach is possible only in restricted domains. Rao and Georgeff [1995] proposes that:

> "*(...) a wide class of other real-time application domains exhibit a number of important characteristics:*
>
> 1. *At any instant of time, there are potentially many different ways in which the environment can evolve (formally, the environment is nondeterministic)*;
> 2. *At any instant of time, there are potentially many different actions or procedures the system can execute (formally, the system itself is nondeterministic)*;
> 3. *At any instant of time, there are potentially many different objectives that the system is asked to accomplish*;
> 4. *The actions or procedures that (best) achieve the various objectives are dependent on the state of the environment (context) and are independent of the internal state of the system*;
> 5. *The environment can only be sensed locally*;
> 6. *The rate at which computations and actions can be carried out is within reasonable bounds to the rate at which the environment evolves.*"

Considering these characteristics, this approach is restricted to domains where the degree of dynamism is relatively low, that is, the environment is relatively stable. In addition, it requires the agent to have enough information about the world, for both the current situation and what lays ahead, and the number of choices to be limited and acceptably low, otherwise the computation of the tree is compromised.

For example, this behaviour is feasible for the Rubik's Cube resolution task in the illustrative scenario. As the cube is a static element – i.e. colours are not changing or pieces falling apart – it is possible to mentally conceive the complete set of moves before physically moving the pieces.

Plan on-demand behaviour. In this deliberative behaviour, the basic actions of a plan are formed and executed, and the outstanding sub-goals are resolved on-demand. Also referred to as *think-act behaviour* in BDI literature, several works, such as Rao and Georgeff [1995], Inverno et al. [2004] and Dastani et al. [2003], describe different techniques to implement this behaviour.

In this model, the agent does not consider all possible actions ahead; just the ones they can reason about with their limited information and bounded resources. This is clear in the illustrative example, where the tourist does not know about the road block when he is still at (a). Thus, the agent thinks only on what it can do immediately ahead, and acts by executing the next possible basic action. Once a new sub-goal is reached, the agent re-considers the possible courses of action to satisfy that sub-goal. The process repeats until there are no sub-goals to pursue – presumably, because the top goal has been achieved or due to either an exception or being abandoned.

Rao and Georgeff [1995] proposed the basic structures for plan revision and reconsideration of actions for the simple BDI-model deliberation algorithm. It concludes that the essential characteristics which contribute to the success of think-act behaviour are:

- "*The ability to construct plans that can react to specific situations, can be invoked based on their purpose, and are sensitive to the context of their invocation facilitates modular and incremental development. It allows users to concentrate on writing plans for a subset of essential situations and construct plans for more specific situations as they debug the system. As plans are invoked either in response to particular situations or based on their purpose, the incremental addition of plans does not require modification to other existing plans.*

- *The balance between reactive and goal-directed behaviour is achieved by committing to plans and periodically reconsidering such committed plans.*

- *The high-level representational and programming language has meant that end-users can encode their knowledge directly in terms of basic mental attitudes without needing to master the programming constructs of a low-level language.(...)*".

Rao [1996] suggests that this model promotes the balance between *proactiveness* and *reactiveness*; where the planning process can think ahead a number of interactions by analysing the tail part of the plan. At the same time, it can re-consider the current basic actions, the remaining sub-goals, or the plan execution itself in reaction to changes of circumstances. This requires that the inference system contains the structures to observe the environment and revise the plan execution process.

However, a number of current BDI implementations work by forming *one-step ahead* solutions; that is, they plan for the immediate requirement considering the current conditions. As the next step will be calculated on the current situation, this approach leads to more *reactive behaviour*. It can be more resource efficient, however as the planning process does not waste cycles calculating unfeasible possibilities while looking for the best match. Nonetheless, Sardina et al. [2006] suggests that:

> "*BDI agent-oriented systems are extremely flexible and responsive to the environment, and as a result, well suited for complex applications with real-time reasoning and control requirements. However, a limitation of these systems is that they normally do not lookahead for the complete solution, thus they cannot be considered as planning in the traditional sense; execution is based on a user-provided plan library to achieve goals. BDI frameworks rely entirely on context sensitive subgoal expansion, acting as they go.*"

In some circumstances, planning ahead is clearly desirable, or even mandatory, for guaranteeing goal achievability and avoiding undesired situations. As stated in Sardina et al. [2006], "*this is the case when (a) important resources may be used in taking actions that do not lead to a successful outcome; (b) actions are not always reversible and may lead to states from which there is no successful outcome; (c) execution of actions take substantially longer than thinking (or planning); and (d) actions have side effects which are undesirable if they turn out not to be useful*".

To remedy this situation, current implementations rely on techniques for *reconsideration* and *backtracking*[4]! As the agent is executing its immediate actions, which count on incomplete information, the plans may lead to dead-ends and exceptions. As detailed below, when facing the unexpected, the agent must behave coherently by revising what went wrong, and backtracking to the

[4]*Backtracking*: this work does not envision backtracking as a possibility, however; I highlight that the execution of basic actions can imply in physical actions – i.e. actions upon the environment – that cannot be rolled back once executed.

previous selection point. Backtracking is not always possible with physical attitudes, as some actions that changed the world situation cannot be undone. Therefore, although we ideally want to plan with the maximum information possible, sometimes we have to assume incomplete plans and be prepared to react to unexpected events.

In this work I am proposing an extension of this model, where environmental information helps to decide emphwhen to reconsider the current elements. I suggest that this technique leads to better deliberation performance.

3.2.3 Plan execution

The process of plan execution is dependent on the agent's planning behaviour. For the *plan ahead behaviour* described above, the process is trivial: execute the inferred actions sequentially. For the *plan on-demand behaviour*, the process involves an extra level of complication as sub-planning levels interleave with plan execution. Nonetheless, the issues of plan execution are the same: capture *exceptions*; analyse them; *recover* when possible, and either *revise* the plan, or *reconsider* the intention.

There is a dilemma however, as described in Rao and Georgeff [1995]: "*reconsidering the choice of action at each step is potentially too expensive and the chosen action possibly invalid, whereas unconditional commitment to the chosen course of action can result in the system failing to achieve its objectives.*" That work proposes that, assuming potentially significant changes can be determined instantaneously, it is possible to limit the frequency of reconsideration and achieve an appropriate balance between too much reconsideration and not enough. The parameters and results for that adjusting were not explored in that work. Schut et al. [2004] analyses the issue and presents a model for BDI agents that are able to determine their own intention reconsideration strategy based on future goals and arising opportunities. It concludes that:

> "*Firstly, an agent's effectiveness increases as its reasoning mechanism is more flexible. Secondly, when the environment's rate of change increases, the level of commitment decreases. This corresponds to the intuition that intentions are more liable to reconsideration when the environment changes fast. Finally, the experiments showed that as planning takes longer, the level of commitment decreases. This can be explained as follows: when it takes longer to plan, the probability that the environment changes during planning increases. In order to cope with this, one needs to replan sooner rather than later.*"

The illustrative scenario clearly shows this somewhat obvious conclusion. The tourist does not reconsider his intention to arrive at (d) using path (a)-right/(b)-left until he reaches the dead-end, showing the expected level of commitment in a rational agent. When he has to reconsider his intention and backtrack to (b) in order to find an alternative, he immediately considers (b)-right as the next possibility, based on utility maximisation, as explained above. However, what if this is also a dead-end? In this case, one assumes that the tourist will start to rely less on his commitments and reconsider the track he has chosen, possibly by collecting more information, such as asking other people about road blocks ahead.

3.2.4 Adaptation and Learning

The agent must be able to adapt its deliberative behaviour to improve performance. Thus, adaptation is a consequence, not an action. That is, the adaptation actions take place either in response to exceptions, generated by unexpected events during plan execution, or as part of the conflict resolution process. One could say, although possibly controversially, that adaptation is the very basis of intelligence. Its stimuli come from exposure to *diversity*, which is a consequence of a dynamic environment. Of course, it presupposes that agents provide the structures to support adaptation; moreover, it requires perceptors to sense changes in the environment and construct a rich representation of the model of the world.

Agents can weigh past experience, cost of actions, and the probability that a sequence of actions leads to success. For example, the tourist concludes that the chances and costs to go from (a) to (d) are the same, so taking either left or right is a guess. However, at (b) the decision is clearly based on costs. As he has no further information, the probabilities to arrive at (d) via either of the bifurcations is always even. Nonetheless, as soon as he learns about the dead-end, he adjusts his knowledge base. Thus, next time he confronts that situation, he will take the choice that maximises or at least, does not cancel – the probability of success.

Therefore, essentially changes can occur in two ways: either *self-modification*, or adaptive changes to the deliberation cycle, or *learning*, as an accumulation of knowledge. As mentioned, it assumes that the agent is provided with access to the structures that compose the deliberation process, that is, to the meta-reasoner. This structure contains the set of meta-reasoning information and the elements about how to reason. Hence, intelligent agents should be able to adapt not only their knowledge – i.e. their program or representation of the world, such as belief bases and plan rules – but also the behaviour of the deliberation process.

A Conceptual Model for the Adaptation Process

This section provides an overview on the concepts behind adaptation in computational systems. For practical reasons, I am considering the problem of adapting computation processes.

For the sake of this discussion, let us consider an abstract architecture for intelligent agents, as proposed in Wooldridge [1999]. The agent contains a set of possible states: a current state, a set of final states, a set of transition rules, and the state-transition function, which is an operation between the current state and the transition rules generating a new state. Intuitively, one can relate these definitions with the terminology commonly used in software agent literature: the current state relates to the notion of *belief base* and *goals base*; the transition rules relate to the notion of *practical reasoning rules* and meta-reasoning rules, associated to the deliberation process; and the state-transition function relates to the notion of deliberation cycle.

In this representation, *adaptation* is the act of changing either the current state or the transition rules by means of a *metareasoning structure*, parallel to the deliberation cycle. The *adaptation act* can be described as the modification of the set of transition rules such that before the adaptation the transition for a state S results into the state S' and after the adaptation the transition for a similar state S results into the state S''. In addition, it should be possible to say that S'' is better than S' by some evaluation criteria; that is, the adaptation act changed the way the application deliberates over a same state, supposedly for the better.

I highlight that there are three levels of adaptation considered in this work: (i)*adaptation of the perception*, or how the agent perceives and represents the world; (ii)*adaptation of the program*, or how the agent processes the elements of the world, and (iii)*adaptation of the deliberative behaviour*, or how the agent processes the program, which processes elements of the world. For the first case, it means to change elements of the model of the world (i.e. beliefs) to influence the deliberation process. In case of a plan review, for example, you can change costs or prioritisation rules as part of the review process and re-execute the plan (when feasible, of course). Nonetheless, the adaptation act is computationally simpler than changing the plan or the deliberation function.

Requirements for Adaptation

There are several methods for implementing adaptation. The simple approach is to directly learn from environmental exceptions. A more complex approach is to learn based on indirect facts, such as if *a* causes *b* and *b* causes *c*, then *a* causes *c* indirectly. Introspection, or learning how the agent understands and reasons about the facts, is an even more complex approach. Regardless of the adaptation process' sophistication, three qualifications are required for its practical implementation, as identified by Schmill et al. [2007]: need for information, ability to make recommendations, and ability to monitor performance improvement. I shall detail how to address these issues later in this chapter.

For example, the tourist learns that (a)-right is not a viable option. This is accumulated knowledge that can be represented in the belief base or as a reviewed plan rule. He can also learn indirectly that there is a good reason why there is another sign at (b) pointing to a longer path. That is, he can learn a generic rule of deliberation that describes this condition – follow the signs when they appear. If the tourist has to make a decision between several possible options in a very restricted amount of time, then the deliberation behaviour itself is adjusted, by including a rule that says "if the number of possible options is high and the current position is near out the context, then take the first option.

3.2.5 Desiderata

In this work, I propose a different approach for the planning process that further extends the *plan on-demand behaviour*. It requires that the agent system is equipped with structures to observe and react to changes within the situation. It must also support concurrent intentions and, consequently, parallel plan resolution. Thus, the planning system must implement intention reconsideration, checkpointing of the deliberation process, and so on. To support these features, the agent's knowledge bases must contain the model of the world and, in addition, information about how to resolve conflicting and unexpected situations, such as resource and goal conflicts, goal prioritisation, and cost of actions for qualitative decision taking.

In what follows, I demonstrate how agent-based applications can support the development of enhanced mobile personal assistants by exploiting environmental information to improve the application's performance. This information is used to decide *when* and *how* to reconsider the deliberation process. In addition, it promotes stability and it reduces the development effort, as the agent's programmer does not need to consider every possible reconsideration strategy *a priori*. I aim to build an open, extensible and re-usable infrastructure for an extended BDI-model inference system. Next, I introduce the syntax to describe the elements of this model.

3.3 Syntax

To introduce a syntax to represent the elements of the agent structure, I refer to the definitions introduced in Hindriks [2001].

Comments on Notation and Terminology

This work uses a simple mathematical notation to make ideas precise. I apply the formalism of discrete maths; a basic understanding of sets and first-order logic should be sufficient to make sense of the various definitions presented.

Notation. If $\mathcal{S}$ is an arbitrary set, then $\wp(\mathcal{S})$ is the powerset of $\mathcal{S}$, and $\mathcal{S}*$ is the set of sequences of elements in $\mathcal{S}$. $false$ is a defined term; the symbol $\neg$ is used for logical negation, so $\neg p$ reads "not p". $\wedge$ is used for conjunction, so $p \wedge q$ reads "p and q". $\vee$ is used for disjunction, so $p \vee q$ reads "p or q"; implication $\rightarrow$, and existential quantification $\exists$ and $\forall$ are defined as usual (from Mendelson [1979]).

Terminology. A formula of the form $p(t)$ is called an *atom*; the set of atoms is denoted as At. The notions of *free* and *bound variables* are defined as usual (*viz.* Lloyd [1987]). A variable X is *bound* if it occurs within the scope of some quantifier $\forall X$, otherwise it is *free*. The set of free variables in an expression E is denoted as $\mathsf{Free}(E)$. A *ground atom* is an atom without occurrences of free variables; a *ground term* is a term without occurrences of free variables.

Next I apply these notation and terminology to introduce the basic components of the proposed language.

3.3.1 Beliefs

Beliefs describe the information the agent has about the world. They are stored in the so-called *belief base*; a structure containing the information that the agent both perceives from and infers about the external environment, and internal conditions.

I adopt the definitions proposed in [Hindriks, 2001, page 12-13]. The beliefs of an agent are formulas from some first order language $\mathcal{L}$, built from a signature $\Sigma = \langle \mathsf{Pred}, \mathsf{Func} \rangle$ of predicate symbols Pred and function symbols Func with associated arities and a countably infinite set of variables Var with typical elements $x, y, x, \ldots, x_1, \ldots$. Let Term denote the set of all terms where: all constants are terms; all variables in Var are terms, and any expression $f(t_1, \ldots, t_n) \in \mathsf{Func}$ with $n \geq 1$ arguments, where each argument t_i is a term and f is a function symbol of arity n, is a term.

Definition 1. (first order formulas)
The set of first order formulas $\mathcal{L}$ is inductively defined by:

- *if $p \in \mathsf{Pred}$ of arity n and $t_1, \ldots, t_n \in Term$, then $p(t_1, \ldots, t_n) \in \mathcal{L}$.*
- *if $t_1, t_2 \in \mathsf{Term}$, then $t_1 = t_2 \in \mathcal{L}$.*
- *if $\varphi \in \mathcal{L}$, then $\neg\varphi \in \mathcal{L}$.*
- *if $\varphi \in \mathcal{L}$ and $\psi \in \mathcal{L}$, then $\varphi \wedge \psi \in \mathcal{L}$.*
- *if $\varphi \in \mathcal{L}$ and $x \in \mathsf{Var}$, then $\forall x(\varphi) \in \mathcal{L}$.*
- *if $\varphi \in \mathcal{L}$ and $x \in \mathsf{Var}$, then $\exists x(\varphi) \in \mathcal{L}$.*
- *the set of conditions $C \in \mathcal{L}$, to be introduced below, is a special element of $\mathcal{L}$.*
- *the desire's element redo $\in \mathcal{L}$, to be introduced below, is also a special element of $\mathcal{L}$.*

The *belief base* is defined in terms of the underlying knowledge representation language as described below.

Definition 2. (belief base)
A belief base $\mathcal{B}$ is a consistent set of closed formula and $\mathcal{B} \nvDash (p \wedge \neg p)$[5]*.*

3.3.2 Goals, Desires, and Intentions

Goals describe the state of affairs that an agent would like to achieve. Goals can be declarative or descriptive. Declarative goals do not specify how the agent should go about realising them, while descriptive goals specify a recipe for actions that the agent intends to follow. This second kind of goal is also called *goals to-do*. Bratman [1987] analysed these structures in terms of plans. From a computational perspective, it is quite natural to focus on the procedural type of goal because it can be described as "*plans to reach a desired state*".

For the proposed model, I opted to differentiate two types of goals: desires and intentions. *Desires* describe the goals that the agent wants to pursue when a given condition is achieved. They exist in a *desire base*, which can hold potentially inconsistent goals, which the deliberation cycle will select when the condition is favourable. *Intentions*, on the other hand, describe the

[5]*Entailment*: where the entailment relation $\nvDash$ is the usual relation in classical logic.

goals that the agent has adopted and is committed to reach. These goals must be consistent. Where there are conflicting options during the intention deliberation, conflict resolution rules resolve the situation.

This differentiation is a positive feature for a BDI-based model, although. some agent literature authors do not describe *desires*. Cohen and Levesque [1990] describes *intentions* in terms of beliefs and goals, which are both primitive logic. Rao and Georgeff [1991] describe intentions as primitives; although in a later paper, Rao and Georgeff [1998] replaces the goal operator with a desire operator.

Notice that not having an explicit *desires* representation prevents the programmer from representing the "*long lasting goals that the agent wants to achieve when the situation is favourable*"; resulting in agent languages considering agent programs as closed circuits. That is, the applications exist for a very specific purpose represented by a set of goals stored in a goal base, which are activated during agent initialisation; a behaviour the literature for example, *viz.* Wooldridge and Jennings [1995] and Dastani et al. [2003] sometimes called the "*mentalistic agent*" [6].

Therefore, the proposed model must support applications that can also run in the background. In this model, the applications are designed with a set of desires that are adopted dynamically when their conditions hold with the model of the world. These desires remain dormant, or inactive, while the conditions are not adequate for their adoption. The deliberation cycle decides which goal to adopt, based on the context and a set of conflict resolution rules. It is also possible to conceive programming constructs that allow the application to assert and retract desires at run time. I propose that this feature facilitates the development of practical applications, providing the model with flexibility and extensibility.

Table 3.1 clarifies the relationship between the terminology adopted in this work and that used in other work. *Goals* are represented as formulas in the set of goals $\mathsf{Goal} \subset \mathcal{L}$.

Desires represent the set of goals that the agent wants to achieve in a given situation. Thus, a desire is a goal associated with a condition. While the set of desires can potentially be logically inconsistent, the desire entries are stable in the sense that they are not modified by deliberation operations.

Definition 3. (desire)
A desire is the tuple $\langle \varphi, C, redo \rangle$, where $\varphi \in \mathsf{Goal}$, $C \in \mathcal{L}$ is the commitment

[6] *Mentalistic Agents*: I do not dispute the computation validity of this model, which is of course correct; however, I question its support to practical application development, where *flexibility* and *extensibility* are desirable features.

Element	Representation	Description
Goals	$\mathsf{Goal} = \{\varphi_1, \ldots, \varphi_w\}$, where $\varphi_i \in \mathcal{L}$ and $1 \leq i \leq w$	logically consistent
Desires	$\mathcal{D} = \{\langle \varphi_1, C_1, redo_1\rangle, \ldots, \langle \varphi_x, C_x, redo_x\rangle\}$, where $\varphi_i \in \mathsf{Goal}$ and $C_i \in \mathcal{L}$ and $1 \leq i \leq x$	the set of desires is potentially logically inconsistent, desires are stable
Intentions	$\mathcal{I} = \{\langle \varphi_1, C_1\rangle, \ldots, \langle \varphi_y, C_y\rangle\}$, where $\mathcal{D} \models \varphi_i$ and $\mathcal{B} \models C_i$ and $1 \leq i \leq y$	the set of intentions must be logically consistent, intentions are transient
Plans	$\mathcal{P}=\{\langle st_1, \varphi_1, \pi_1, \Theta_i\rangle$, ..., $\langle st_z, \varphi_z, \pi_z, \Theta_z\rangle\}$, where the deliberation of $\mathcal{I} \models \varphi_i$ implies the execution of plan $\pi_i \in \mathsf{Plan}$, $st_i \in \{run, pause\}$ is the plan deliberation status, Θ_i is the unification stack, and $1 \leq i \leq z$	each plan must be resource consistent, plans are transient

Table 3.1: Terminology for Goals, Desires, Intentions, and Plans

condition*, and* $redo \in \mathcal{L}$ *is the recurrent flag, which can be true or* $false$*.*

It means that if the desire's condition holds with the model of the world, then the goal φ is adopted. If *redo* is true, then the desire is permanent; that is, not revoked after the goal is achieved. In practice, they are used for long-term desires. If *redo* is false, then the desire is retracted after the goal is adopted. This latter format is used for short-term (i.e. one-shot) desires.

The concept of *commitment condition* is part of the requirements for operating in dynamic environments. Rao [1996] says it is part of the dynamic constraint; with the agent having no direct control over its beliefs and desires, there is no way it can adopt or effectively realise a commitment strategy over the adoption and commitment attitudes.

However, an agent can choose what to do with its desires. The commitment condition provides the information to evaluate whether it should pursue the desire in the current situation; such that the agent should commit to the goal φ if the condition C holds with the belief base. In addition, the agent should insist with φ while the condition C is true. Desires are stored in a *desire base*, as defined below.

Definition 4. (desire base)
$\mathcal{D} = \{d_1, \ldots, d_n\}$ *is a* desire base*, where* d_j *is a desire and* $1 \leq j \leq n$*.*

Once a desire is selected, pursuing the goal φ in the condition $C\theta$[7] becomes an intention. It is important to highlight that committing to goals lends a certain sense of stability to an agent's reasoning process.

Therefore, intentions are the consistent set of chosen goals deliberated when the conditions for given desires held with the model of the world. These structures are more dynamic than desires as they can be reconsidered in reaction to environmental changes.

Definition 5. (intention)
An intention is the tuple $\langle \varphi, C \rangle$, where $\varphi \in Goal$ and $C \in \mathcal{L}$ is the termination condition*.*

In addition, deliberating a new intention implies deliberating the plans to resolve the intended goal. *Plans* are resource-consistent structures that determine the set of actions the agent must execute to achieve a desired goal. More dynamic than intentions, their steps can be revised in reaction to either environmental changes or execution exceptions.

As mentioned, the agent persists in reaching φ while the condition C is true. This behaviour results in computational effort savings, and hence better overall performance. In the next section, I introduce agent structures to optimise the process that verifies conditions against the environment; namely, the context observer.

Intentions are stored in a *intention base* defined below.

Definition 6. (intention base)
$\mathcal{I} = \{i_1, \ldots, i_n\}$ *is the* intention base *where i_j is an intention and $1 \leq j \leq n$.*

3.3.3 Basic Actions and Capabilities

Basic actions specify an agent's capacity to modify its environment:

Definition 7. (basic actions and abstract plans)
Let Asym *be a set of action symbols with typical elements $a, a', \ldots, b, \ldots$ each with a given arity. Let* AAsym *be a set of abstract action symbols with typical elements $x, x', \ldots, y, \ldots$ each with a given arity, where* $\mathsf{Asym} \cap \mathsf{AAsym} = \emptyset$*. Then $a(t_1, \ldots, t_n)$ is an* basic action *for any action symbol $a \in \mathsf{Asym}$ and terms $t_1, \ldots, t_n \in \mathsf{Term}$ where* n *is the arity of a. Similarly, $x(t_1, \ldots, t_n)$ is an abstract plan for any action symbol $x \in \mathsf{AAsym}$ and terms $t_1, \ldots, t_m \in \mathsf{Term}$ where* m *is the arity of x. The set of basic actions is denoted by* BAct*. The set of abstract plans is denoted by* $\mathsf{AbstractPlan}$

[7] *Condition*: where θ contains the unification of the resolved variables, as I shall explain later in this work.

Basic actions can be performed only if an *enabling condition* holds with the model of the world; such as a condition checking whether the environment contains a given resource that supports the execution. These are the *preconditions* of the action.

After the action has been executed, it implies some effect over the model of the world. For example, an action to move north implies a change of agent location northwards. These are an action's *postconditions.*

Therefore, an agent's capabilities describe the *preconditions* where certain actions can take place and the *postconditions* of the expected outcome of the execution. This structure is defined below.

Definition 8. (capability)
A capability is the tuple $\langle C, \mathsf{Act}, C' \rangle$*, where* $C \in \mathcal{L}$ *is the precondition,* φ *the goal, and* $\mathsf{Act} \in \mathsf{Bact}$ *is the sequence of actions, and* $C' \in \mathcal{L}$ *is the postcondition* .

Capabilities are stored in a *capability base*, defined below.

Definition 9. (capability base)
$\mathcal{A} = \{a_1, \ldots, a_i, \ldots, a_n\}$ *is the* capability base *where* a_i *is a capability.*

The circumstances and operations to execute capabilities are introduced in the next section.

3.3.4 Plans and Practical Reasoning Rules

A plan is a sequence of *basic actions* and *abstract plans* formed from the agent's deliberation about its current situation using practical reasoning rules. Once executed, the actions result in either the agent's belief base or the physical environment changing, depending on the agent language. *Abstract plans* work as an abstraction mechanism like procedures in imperative programming. If a plan consists of an abstract plan, this abstract plan could be transformed into basic actions through reasoning rules. I adopt the definition of plan from [van Riemsdijk and Meyer, 2006, page 8] as:

Definition 10. (plan)
Let us consider a given element $a \in \mathsf{BAct}$*, and an element* $p \in \mathsf{AbstractPlan}$*; let* $c \in (\mathsf{BAct} \cup \mathsf{AbstractPlan})$*; the set of plans* Plan *with typical element* π *is defined as:*

$$\pi ::= a \mid p \mid c; \pi'$$

As mentioned, the agent language provides semantics for composing plans from the practical reasoning rules; a process called plan formation. Once composed, plans are executed by both (i) processing the basic actions at the head of the plan, thus modifying the belief base, and (ii) deliberating about the abstract plans at its tail.

Practical reasoning rules supply the agent with a facility to manipulate goals. They build a plan library from which an agent can retrieve plans for achieving an achievement goal, and provide the means to revise and monitor the agent's goals.

These rules provide the reasoning capabilities for agents to reflect on their goals. Informally, they can be described as follows: from the goal to achieve φ and the belief that the plan π is sufficient to achieve φ, the agent concludes that it is reasonable to adopt π. This type of reasoning is also called *means-end reasoning.*

In addition, the agents may also have to reconsider the adopted plans. This reasoning follows a pattern similar to that stated above; a commitment to a goal φ leads to the adoption of the plan π, however during execution, a belief σ activates that the situation requires replacing π with an alternative course of action π'. This scenario provides the agent with reasons to conclude that it should adopt π' and remove π as its current course of action. I classify both types of reasoning under the label of *practical reasoning*, although I split the explanation between *plan selection rules* and *plan revision rules.*

Plan selection rules allow the programmer to specify which plan an agent should execute for a given goal. They are of the form $\varphi \leftarrow C \mid \pi$, which means that the deliberation cycle can select the plan π to achieve the goal φ if and only if the condition C holds with the model of the world (i.e. the belief base).

Similarly, *plan revision rules* allow the programmer to specify rules to revise its plans at run-time. They are of the form $\pi \leftarrow C \mid \pi'$, which express that if the agent has the plan π and believes the condition C to be true, then it may replace the plan π by the plan π'. These rules are stored in two sets of rules that comprise the agent's know-how, based on [van Riemsdijk and Meyer, 2006, page 9]:

Definition 11. (practical reasoning rules)
The practical reasoning rules Γ *contains the set of plan selection rules* PS *and the set of plan revision rules* PR *defined as follows:*

> $PS = \{\varphi_1 \leftarrow C_1 \mid \pi_1, \ldots, \varphi_i \leftarrow C_i \mid \pi_i, \ldots, \varphi_n \leftarrow C_n \mid \pi_n\}$*, such that* $\varphi_i \in \mathsf{Goal}$*,* $C \in \mathcal{L}$*, and* $\pi_i \in \mathsf{Plan}$
> $PR = \{\pi_1 \leftarrow C_1 \mid \pi'_1, \ldots, \pi_j \leftarrow C_j \mid \pi'_j, \ldots, \pi_m \leftarrow C_m \mid \pi'_m\}$*, such that* $C_j \in \mathcal{L}$*,* $\pi_j, \pi'_j \in Plan$

In this work, I propose that the deliberation module must support executing multiple plan formation processes in parallel. This feature allows the agent to resolve multiple goals at same time, as long as they are not conflicting.

Such a feature requires the structure to coordinate parallel planning. In addition, it must be able to detect and resolve conflicts, support prioritisation, and pause, resume, or stop any of the planning processes individually. These features are essential to support dynamic, complex environments like mobile services.

3.3.5 Planning Threads

Planning threads represent the deliberation of individual intentions. Each thread contains the information required to form and execute the plan for the intention's goal; thus, is self-contained. This representation provides a mechanism to implement intention reconsideration. This structure is defined below.

Definition 12. (planning thread)
A planning thread *is the tuple* $\langle st, \varphi, \pi, \Theta \rangle$ *where* st *is the operation status as a value in the set* $\{run, pause\}$*,* $\varphi \in \mathsf{Goal}$ *is the committed goal,* $\pi \in \mathsf{Plan}$ *is the current plan, and* Θ *is the local unification stack.*

The operation status provides a control over the thread status. The unification stack Θ contains the set of substitutions used to resolve the terms. Resolving the terms' free variables generated a set of substitutions θ, where $t\theta$ denotes the expression where all free variables that were substituted while resolving term t. The unification stack contains the pairs $\langle t, \theta \rangle$ in a sequential order following a "last in, first out" arrangement. Thus, it is possible to review what variable substitutions are associated with past operations. This information is used, for example, to resume the execution of a plan thread when the current bindings must be reconsidered. I shall explain this process later on in this chapter.

It is possible to control the plan formation process by storing this information in individual structures. These elements are stored in a *plan base*, which is part of the agent configuration. This structure is defined below.

Definition 13. (plan base)
$\mathcal{P} = \{pt_1, \ldots, pt_i, \ldots, pt_n\}$ *is the* plan base *where* pt_i *is a planning thread.*

As the planning thread is associated with the intention to achieve a goal φ, it presupposes that the goal still exists in the desire and intention bases. The operational constraint is that if either the desire or the intention has been retracted, then the planning threads must also be retracted. It is worth highlighting that the intention deliberation process resolves the free variables in φ against the current belief base configuration and asserts a new intention with $\langle \varphi\theta_i, C, redo\rangle$, where $\mathcal{B}_i \models \varphi\theta_i$. Described in Section 3.4.1, this process means that the intention base can contain several variations of the desired goal, with different unifications. The constraints for planning thread existence are defined below.

Constraint 1. (constraints in planning thread existence)

1. $\forall\langle X, \varphi\theta_i, Y, Z\rangle \in \mathcal{P} : \exists\langle \varphi, C, redo\rangle \in \mathcal{D}$
2. $\forall\langle X, \varphi\theta_i, Y, Z\rangle \in \mathcal{P} : \exists\langle \varphi\theta_i, C\rangle \in \mathcal{I}$

The first formula says that there should not be a planning thread for any resolved form $\varphi\theta_i$ of a desired goal φ if the agent no longer desires that goal. Similarly, the second formula says that there should not be a planning thread for the resolved goal $\varphi\theta_i$ if the agent no longer intends to achieve it. I propose that the implementation must provide programming constructs to enforce these constraints.

3.3.6 Agent

An agent is an entity that represents its environment by means of its *beliefs*; controls this environment by means of *basic actions*, and resolves its *goals* using *practical reasoning rules*. These elements change states during the agent's operations by means of the *transition rules*, with each transition performed in one computational interaction.

Beliefs, desires, intentions, and plans compose the agent's *mental state*, which is part of its configuration. This structure is defined below.

Definition 14. (agent configuration)
The configuration of an agent is the tuple $\langle\mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P}\rangle$*, where* $\mathcal{B}$ *is the* belief base*,* $\mathcal{D}$ *is the* desire base*,* $\mathcal{I}$ *is the* intention base*, and* $\mathcal{P}$ *is the* plan base*.*

Hence, an agent is defined as below.

Definition 15. (agent)
An agent *is the tuple* $\langle i, \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P}, \mathcal{A}, \Gamma\rangle$*, where* i *is the agent identifier,* $\mathcal{B}$ *is the* belief base*;* $\mathcal{D}$ *is the* desire base*;* $\mathcal{I}$ *is the* intention base*;* $\mathcal{P}$ *is the* plan base*;* $\mathcal{A}$ *is the capability base, and;* Γ *is the set of practical reasoning rules.*

The agent is animated by the *deliberation process*, which contains a set of deliberation functions. These functions provide the building blocks for the deliberative behaviour. To program an agent means to:

1. specify its initial mental state, including its initial beliefs and desires;
2. specify the basic actions that define the expertise of the agent, and;
3. write the set of planning rules, which define the agent's know-how.

The deliberation cycle combines entries of the desire base with the model of the world, committing to goals (intentions) that are processed through plans. An agent can have more than one goal at the same time. Each thread for goal resolution is called a *planning thread*, which exist in the *plan base*. These structures are defined in the next sub-section.

In this thesis, I am considering the practical aspects of planning and plan formation processes. I focus on the development of a planning structure that is reactive to events from the environment. I claim that this extension is required to support the objectives of the design. It provides impromptu reconsideration of in-processing elements to take advantage of events from the window of opportunity, to maintain the application's coherence amidst environmental instabilities, and to execute with constrained resources. These issues are related to the fundamental issues in highly dynamic environment, introduced in section 3.2.

Next, I introduce the operations of the deliberation process.

3.4 Operations

In this section, I detail the operations of the deliberation process. This process animates the agent *via* state transitions based on the information about the environment and the elements of the agent program; that is, the programmed desires and planning rules. As the situation evolves, the deliberation cycle infers new intentions and composes the plans to resolve these intentions based on the set of practical reasoning rules. These processes are called intention deliberation and plan formation, respectively.

Several literature works introduce the semantics for the agent's operations in terms of transition rules. For example, Hindriks [2001] for 3APL, Dastani et al. [2007] for 2APL, and Inverno and Luck [1998] for AgentSpeak(L). In this section, I introduce the operations of a deliberation system that is extended with a module to observe the context in parallel to the planning module. This new model equips the application to adapt and reconsider the current intentions in response to environmental events. The requirements for this model are:

- *an ability to adapt its deliberation* in response to unforeseen changes in context;

- *an ability to keep the application coherent and consistent* by sustaining means-end coherence throughout plans and being able to reason multiple, conflicting options, and;
- *an ability to pause, resume, stop individual intention execution threads* as a method for plan revision and adaptation; an ability essential for coping with simultaneous intention processing in dynamic environments.

I am extending the existing definitions and propose an operation model that provides support for these requirements; presenting the operations of the deliberation cycle in terms of transition rules. These rules describe how the agent deliberates about new intentions and are plan based.

Next, I introduce the notations used in this presentation.

Comments on Notation and Terminology

I describe the semantics of the operations by means of a *transition system*, as described in Plotkin [1981]:

> "*(...) the operations the system can perform – whether internally or interactively with some supersystem or the outside world. For in our discrete (digital) computer systems behaviour consists of elementary steps which are occurrences of operations. Such elementary steps are called here* transitions. *Thus a transition steps from one configuration to another and as a first idea we take it to be a binary relation between configurations.*"

The derivation rules are called *transition rules*, and each transition corresponds to one single computation step. A set of transition rules can be viewed as an inductive definition of a transition relation $\rightarrow$. It defines the smallest relation that contains all the axioms of the system and all conclusions that are derivable using these axioms and transition rules. The general format of a transition rule is presented below.

$$\frac{c_1 \wedge \cdots \wedge c_n}{C_{concl} \rightarrow C'_{concl}}$$

It means that a set of conditions, or *premises*, $c_1 \wedge \cdots \wedge c_n$ implies the transition of a state, or conclusion, $C_{concl} \rightarrow C'_{concl}$. A transition relation is a relation of configurations that describes how the configuration will change in a given condition.

In the proposed model, the agent configuration is described by means of the agent's mental state (*viz.* Definition 14). Hence, the transition rules introduced in this section describe the relationship between environmental conditions and the transition of the agent's mental state.

Substitutions

Let us assume that the implementation provides the logic basis, the structures for knowledge representation, and the mechanism inference. For instance, the proof-of-concept implementation introduced in Chapter 5 uses PROLOG to that end.

Substitutions are an important feature of operation semantics is used to define the parameter mechanism. For detailed information on the terminology and operation semantics, see [Hindriks, 2001, page 34].

A substitution θ of the first order variable is a finite set of pairs of the form $x_i = t_i$, where $t_i \in \mathsf{Term}$ is a term bound to variable $x_i \in \mathsf{Var}$, and $x_i \neq x_j$ for every $i \neq j$, and $x_i \notin \mathsf{Free}(t_j)$, for any i and j.

The simultaneous replacement of expressions bound to a variable for that variable defines the application of a substitution to a syntactic expression. A *ground substitution* θ is a substitution such that for every pair $x = t \in \theta$, $\mathsf{Free}(t) = \emptyset$. The domain of θ, denoted as $dom(\theta)$, is the set of variables x for which θ contains a pair $x = t$.

Bindings represent the substitutions θ that are being computed place in a transition $C_{concl} \rightarrow_\theta C'_{concl}$.

Moreover, if e is any static expression and θ are the substitutions, then $e\theta$ denotes the expression where *all* free variables in e are simultaneously replaced by the substitutions in θ.

A formal definition for the application of substitutions to formulas can be found in Lloyd [1987].

Unification

Unification involves the notion of matching two terms t_1 and t_2 by resolving their free variables. This is the simplest operation unit in a first-order logic system. A formal definition is introduced in Goguen [1989].

Let X be a set of free variables in t_1, and θ be the substitutions; then $t_1\theta$ denotes the expression where all free variables are substituted. For simplicity, let us assume that the unification of two terms t_1 and t_2 happens if the given conditions are true:

- the grounded elements of both terms match;
- the linked variables of t_1 can unify to either grounded elements of t_2 or the values of linked variables of t_2, and;

- the free variables of t_1 unify to grounded elements of t_2 or the values of linked variables in t_2, and vice-versa.

The meta-operator $\mathsf{unify}(t_1, t_2, \theta)$ represents the unification operation. It means that the term t_1 unifies with the term t_2 resulting in the set of substitutions θ.

The language PROLOG, used for the proof-of-concept implementation in Chapter 5, inherently provides this facility. The counting of the number of unifications in an operation provides a simple way to measure its computational performance; a feature I exploit for the case studies in Chapter 6.

Entailment Relation

I apply the entailment relation between the elements of the intention base and goals, and; the elements of the planning thread base and goals in order to define the presence (or absence) of an intention or planning thread committed to reach that goal.

Hence, the notation $\mathcal{I} \vDash \varphi$, where $\varphi \in \mathsf{Goal}$, means that $\exists\langle\psi, C\rangle \in \mathcal{I} : \psi \vDash \varphi$. Consequently, $\mathcal{I} \nvDash \varphi$ means that $\nexists\langle\psi, C\rangle \in \mathcal{I} : \psi \vDash \varphi$.

Similarly, the notation $\mathcal{P} \vDash \varphi$ means that $\exists\langle Status, \psi, Plan, Stack\rangle \in \mathcal{P} : \psi \vDash \varphi$. Similarly, $\mathcal{P} \nvDash \varphi$ means that $\nexists\langle Status, \psi, Plan, Stack\rangle \in \mathcal{P} : \psi \vDash \varphi$.

In addition, I apply two other notations related to entailment relation. $\mathcal{B} \vDash C\theta$ means that the belief base $\mathcal{B}$ satisfies the condition C with the substitutions in the set θ – where C is the condition element from a desire, intention, or plan. That is, all of the elements in the set C hold with the model of the world.

Moreover, the notation $\pi = \pi'\theta$, where $\pi, \pi' \in \mathsf{Plan}$, means that the plan π unifies to the plan π' resulting the substitutions in θ. In the same way, $\varphi \vDash \varphi'\theta$, where $\varphi, \varphi' \in \mathsf{Goal}$, means that the goal φ implies the goal φ' with the substitutions in θ. That is, the elements in φ unify to the elements in φ' resulting the set of substitutions θ.

3.4.1 Intention Deliberation

Intentions are the goals the agent is committed to achieve. The agent can commit to multiple intentions at the same time, based on existing desires and current conditions. The deliberation cycle must provide the facilities to process them simultaneously, which implies that this process must provide methods for detecting conflicts and promoting processing prioritisation to avoid inconsistencies. This notion is intuitive; such that if the agent in Example 1 is at $pos(3)$ and there are two bases at $base(1)$ and $base(5)$, i.e. one at the left and other at the right of the agent. If the program is not equipped to detect conflicts and tries to resolve $goBase(X/1)$, which implies an action to go-left, and $goBase(X/5)$, which implies an action to go-right, the operations will enter a loop.

In this sub-section, I introduce the methods to detect conflicts before introducing the facilities to promote prioritisation, and developing the operations for intention deliberation based on these methods.

Conflict Detection Rules

Conflicts are a by-product of concurrent processing. An agent can hold conflicting desires however, when committing to multiple goals the intentions must remain consistent. In the above example, the agent cannot commit to both goals if the model of the world implies that both goals are conflicting when executed together. The language must provide facilities to represent conflicts between goals. This structure creates a *filter of admissibility* for the further intentions that the agent can adopt, fulfilling one of the requirements for practical reasoning.

In short, the deliberation cycle cannot hold two running planning threads if their basic actions both demand the same resource. Hence, it must: (i) represent conflict rules as part of the belief base and (ii) provide the operations to handle these situations, as described below.

Definition 16. (representation of conflict rules)
The formula $\mathsf{conflict}(A, B) \in \mathcal{L}$ *represents the conflict between the elements* A *and* B*, where these elements can be either goals or basic actions.*

Hence, two types of conflicts need to be represented. First, there are conflicts between basic actions whose rationale is intuitive: conflicting basic actions cannot be executed at the same time to avoid reaching inconsistent states, such as the instance of trying to go-left and go-right at the same time in the above example.

Let us consider the meta-operator $A \nleftrightarrow B$ that reads "*element* A *conflicts with element* B". Let us also consider the meta-operator $index(a, \pi)$ that reads "the basic action a exists in the plan π". I define the operator to detect conflicting plans below.

Definition 17. (operator to detect conflicting plans)

$$\pi_1 \nleftrightarrow \pi_2 \textit{ iff } \exists a, b : \mathsf{index}(a, \pi_1),\ \mathsf{index}(b, \pi_2),\ conflict(a, b) \in \mathcal{B}$$

For instance, suppose there are the plans π_1 whose body is $a; b; \pi_1'$, and π_2 whose body is $c; d; e; f; g; h; \pi_2'$. Also suppose that the belief base contains the rule $conflict(a, g)$. One can then say that π_1 and π_2 are conflicting, regardless of the position of a and g in the bodies of the plans.

I highlight the fact that due to practical simultaneity limits one cannot say that two actions are conflicting only if they are in the same physical position in the body of a plan. For example, the execution of basic actions can take different amounts of time, and it is not possible to say they will execute at the same moment when the sequence is processed.

Conflicts between goals is a higher level representation than the above. Based on experience, we intuitively learn that certain goals should not be pursued at same time. For instance, the tourist in the example in Section 3.2 does not need to start "search the map" and "resolve the Rubik's Cube" at same time to figure out that this is counter-productive. Based on past experience, he already knows that he cannot execute two actions demanding mental attention, thus he prioritises the more urgent one and pauses the other.

Similarly, to realise that two goals are conflicting the agent should not have to deliberate the intentions and form the plans to actually reach the state where the two plans are conflicting. If these rules are represented in the belief base, then it can decide the prioritisation process during intention deliberation. These rules can be either loaded as part of the agent program or learned during operations. For example, in the event of conflicts being detected during the plan execution process, the learning module can infer that executing the related goals would generate a conflict and add this rule to the belief base.

Hence, I extend the Definition 18 to support the representation of conflicting goals, as outlined below.

Definition 18. (operator for conflict detection due to conflicting goals)

$$\varphi_1 \nleftrightarrow \varphi_2 \textit{ iff } conflict(\varphi_1, \varphi_2) \in \mathcal{B}, \textit{ where } \varphi_1, \varphi_2 \in Goal.$$

As it is intuitive, $\varphi_1 \nleftrightarrow \varphi_2$ because the execution of φ_1 leads to the execution of the plan π_1 and the execution of φ_2 leads to the execution of the plan π_2 and $\pi_1 \nleftrightarrow \pi_2$.

Finally, I introduce the constraints for concurrent intention execution below.

Constraint 2. (concurrent intentions constraint)

$$\nexists \langle \varphi_i, \Theta_i \rangle, \langle \varphi_j, \Theta_j \rangle \in \mathcal{I}, \textit{ such that } \varphi_i \nleftrightarrow \varphi_j.$$

$$\nexists \langle run, \varphi_i, \pi_i, \Theta_i \rangle, \langle run, \varphi_j, \pi_j, \Theta_j \rangle \in \mathcal{P}, \textit{ such that } \pi_i \nleftrightarrow \pi_j.$$

Example 3. (invalid concurrent intention processing)

For example, consider the configuration below. The conflict rule says that the agent cannot pursue $goBase(X)$ and $readPamphlet(P)$ concurrently.

$$
\begin{aligned}
\mathcal{B} =& \{base(5), pos(1), conflict(goBase(X), readPamphlet(P))\}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
& \langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =& \{\langle goBase(X/5), \{base(X/5), NOT\ pos(X/5)\}\rangle\}\\
\mathcal{P} =& \{\langle run, goBase(X/5), \pi_i, \Theta\rangle\}
\end{aligned}
$$

Suppose that during the operations the agent asserts the belief $carrying(p)$, which triggers the deliberation of $readPamphlet(P/p)$. If the deliberation cycle does not implement the operations to enforce the Constraint 2, then the operations will result in the configuration below; where the agent is trying to go to base and read the pamphlet at the same time which is not valid!

$$
\begin{aligned}
\mathcal{B} =& \{base(5), pos(1), conflict(goBase(X), readPamphlet(P)), carrying(p)\}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
& \langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =& \{\langle goBase(X/5), \{base(X/5), NOT\ pos(X/5)\}\rangle,\\
& \langle readPamphlet(P/p), \{carrying(P/p)\}\rangle\}\\
\mathcal{P} =& \{\langle run, goBase(X/5), \pi_i, \Theta\rangle,\\
& \langle run, readPamphlet(P/p), \pi_j, \Theta'\rangle\}
\end{aligned}
$$

To avoid processing $goBase(X/5)$ and $readPamphlet(P/p)$ simultaneously, one of the planning threads should be paused or dropped. That configuration would comply with Constraint 2. The question is which planning thread to pause? For that, I introduce the prioritisation process below.

Prioritisation Rules

Now that I have introduced the rules to detect conflicts, the question is what to do with conflicting deliberations? The literature offers several approaches to solve this issue. Some work suggests running the current planning thread to completion before executing the new one, such as the deliberation cycle proposed for 3APL in Hindriks [2001]. However, this solution does not consider the issue of prioritisation: what if the priority of the new goal is higher than the current goal?

Therefore, when facing possible conflicting executions, the agent has to decide which processing to prioritise. I propose a practical solution based on

(i) the representation of the prioritisation rules as part of the belief base, and (ii) the operation rules to prioritise executions; which I introduce below.

Definition 19. (priority rules)
The formula $\mathsf{priority}(\varphi, C, n) \in \mathcal{L}$ *represents a priority rule, where* $\varphi \in \mathsf{Goal}$*, C is a condition, and n is a numerical value that quantifies the priority; the set of priority rules is defined as* $\mathsf{Priority} \subseteq \mathcal{B}$*.*

That is, the entry represents a priority quantifier for a given goal in a specific condition. I introduce the meta-operators $a <_p b$ that reads "*a has a lower priority than b*". The operations are defined below.

Definition 20. (priority verification operation)
Given two goals $\varphi_1, \varphi_2 \in \mathsf{Goal}$ *and the priority rules:*
$\mathsf{priority}(\varphi_1, C_1, n_1), \mathsf{priority}(\varphi_2, C_2, n_2) \in \mathcal{B}$*.*

- $\varphi_1 <_p \varphi_2 \iff n_1 \leq n_2$
- $\varphi_2 <_p \varphi_1 \iff n_1 > n_2$

Next, I introduce the operations for intention deliberation that take advantage of this representation.

Intention Deliberation

Consider three situations for intention deliberation:

1. when there are no conflicts between the desired goal φ and any other goals in the intention base. In this case, the intention is asserted and a new planning thread is created for $\varphi\theta$ in running status.

2. when there is a conflict between φ and any φ' in the intention base, and φ has a lower priority than φ'. In this case, the intention is asserted and a new planning thread is created for $\varphi\theta$ in paused status.

3. when there is a conflict between φ and any φ' in the intention base, and φ has a higher priority than φ'. In this case, there is a goal switch; that is, the existing planning thread for φ' is paused, the intention is asserted, and a new planning thread is created for $\varphi\theta$ in running status.

The operations for each of these situations are detailed in the next paragraphs.

Deliberating intention with no conflicts. The first situation is the simplest. If there are no conflicts between φ and any of the current intentions, the operation just asserts the new intention and create the planning thread for it. These operations are represented in the transition below.

Definition 21. (operation for intention deliberation with no conflicts)
Let φ be a desired goal and θ be a ground substitution such that $dom(\theta) = \mathsf{Free}(\varphi)$.

$$\frac{\langle \varphi, C, redo\rangle \in \mathcal{D},\ \mathcal{B} \vDash C\theta,\ \mathcal{I} \nvDash \varphi\theta,\ \nexists \varphi_i : \mathcal{I} \vDash \varphi_i \wedge (\varphi \nleftrightarrow \varphi_i)}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P}\rangle \rightarrow_\theta \langle \mathcal{B}, \mathcal{D}, \mathcal{I} \cup \{\langle \varphi\theta, C\theta\rangle\}, \mathcal{P} \cup \{\langle run, \varphi\theta, \epsilon, \Theta\rangle\}\rangle}$$

I highlight that the premise is to verify whether the intention base does not imply the processing of $\varphi\theta$. This condition prevents the application reprocessing the deliberation for the same goal. Moreover, it is assumed that the implementation provides the programming constructs to remove the desire in case $redo = false$; the unification stack Θ is initialised with the contents of θ. An example of this operation is presented below.

Example 4. (intention deliberation with no conflicts)

Let us consider the agent configuration below, where no conflicts are represented.

$$\begin{aligned}
\mathcal{B} =& \{carrying(p), pos(1)\}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
& \langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =& \{\langle readPamphlet(P/p), \{carrying(P/p)\}\rangle\}\\
\mathcal{P} =& \{\langle run, readPamphlet(P/p), \pi_3', \Theta\rangle\}\\
\Gamma =& \{\pi_1 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)'',\\
& \pi_2 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)'',\\
& \pi_3 = \text{“}readPamphlet(P) \leftarrow TRUE \mid Read(P), readPamphlet(P)''\}\\
\mathcal{A} =& \{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
& \{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\},\\
& \{\}Read(P)\{\}\}
\end{aligned}$$

If, as part of external action, a new belief is asserted representing $base(5)$, the operations in Definition 21 take place generating the new status below.

$$\begin{aligned}
\mathcal{B} =&\{carrying(p), pos(1), base(5)\}\\
\mathcal{D} =&\{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
&\langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =&\{\langle readPamphlet(P/p), \{carrying(P/p)\}\rangle,\\
&\langle goBase(X/5), \{base(X/5), NOT\ pos(X/5)\}\rangle\\
\mathcal{P} =&\{\langle run, readPamphlet(P/p), \pi', \Theta\rangle,\\
&\langle run, goBase(X/5), \epsilon, \{X/5\}\rangle\}\\
\Gamma =&\{\pi_1 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)'',\\
&\pi_2 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)'',\\
&\pi_3 = \text{“}readPamphlet(P) \leftarrow TRUE \mid Read(P), readPamphlet(P)''\}\\
\mathcal{A} =&\{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
&\{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\},\\
&\{\}Read(P)\{\}\}
\end{aligned}$$

Deliberating intention with conflict and lower priority. In the second situation, if there is a conflict between φ and any φ' in the intention base, and φ has lower priority than φ', then the intention is asserted and a new planning thread is created for $\varphi\theta$ in paused status. The planning thread will remain paused until the condition to resume the deliberation is asserted.

The alternative is to do nothing; that is, not to include the paused intention deliberation into the plan base. The deliberation cycle keeps checking the condition until either (i) it runs out of the scope or (ii) the current deliberation completes; at which time the new processing thread can start.

Notice that both approaches lead to similar results with computational advantages and disadvantages associated with them. Starting a paused intention deliberation for the sake of computation optimisation avoids the deliberation cycle repeatedly trying to process the intention, and failing due to the conflict rule. I opted for this approach, which is represented below.

Definition 22. (operation for intention deliberation with conflict and lower priority)
Let φ be a desired goal and θ be a ground substitution such that $dom(\theta) =$ $\mathsf{Free}(\varphi)$.

$$\frac{\langle \varphi, C, redo\rangle \in \mathcal{D},\ \mathcal{B} \models C\theta,\ \mathcal{I} \not\models \varphi\theta,\ \mathcal{P} \not\models \varphi\theta,\\ \exists \varphi_i : \mathcal{I} \models \varphi_i \wedge (\varphi_i \not\leftrightarrow \varphi) \wedge (\varphi <_p \varphi_i)}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P}\rangle \rightarrow_\theta \langle \mathcal{B}, \mathcal{D}, \mathcal{I} \cup \{\langle \varphi\theta, C\rangle\}, \mathcal{P} \cup \{\langle pause, \varphi\theta, \epsilon, \Theta\rangle\}\rangle}$$

Example 5. (operation for intention deliberation with conflict and lower priority)

Let us consider the following initial configuration.

$$\begin{aligned}
\mathcal{B} =& \{base(5), pos(1), conflict(goBase(X), readPamphlet(P)),\\
& priority(goBase(X), \_, 1), priority(readPamphlet(P), \_, 5), \}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
& \langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =& \{\langle goBase(X/5), \{base(X/5), NOT\ pos(X/5)\}\rangle\}\\
\mathcal{P} =& \{\langle run, goBase(X/5), \pi_1', \Theta\rangle\}\\
\Gamma =& \{\pi_1 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)'',\\
& \pi_2 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)'',\\
& \pi_3 = \text{“}readPamphlet(P) \leftarrow TRUE \mid Read(P), readPamphlet(P)''\}\\
\mathcal{A} =& \{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
& \{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\},\\
& \{\}Read(P)\{\}\}
\end{aligned}$$

If, as part of external action, a new belief is asserted representing carrying(p), the operations in Definition 22 take place, then the situation evolves generating the new status below.

$$\begin{aligned}
\mathcal{B} =& \{base(5), carrying(p), pos(1), conflict(goBase(X), readPamphlet(P)),\\
& priority(goBase(X), \_, 1), priority(readPamphlet(P), \_, 5), \}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
& \langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =& \{\langle goBase(X/5), \{base(X/5), NOT\ pos(X/5)\}\rangle,\\
& \langle readPamphlet(P/p), \{carrying(P/p)\}\rangle\}\\
\mathcal{P} =& \{\langle run, goBase(X/5), \pi_1', \Theta\rangle,\\
& \langle pause, readPamphlet(P/p), \epsilon, \{X/5, P/p\}\rangle\}\\
\Gamma =& \{\pi_1 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)'',\\
& \pi_2 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)'',\\
& \pi_3 = \text{“}readPamphlet(P) \leftarrow TRUE \mid Read(P), readPamphlet(P)''\}\\
\mathcal{A} =& \{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
& \{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\},\\
& \{\}Read(P)\{\}\}
\end{aligned}$$

Deliberating intention with conflicts and goal switching. Finally, if there is a conflict between φ and any φ' in the intention base, and φ has higher priority than φ', then the deliberation cycle switches the processing between the goals, by pausing φ' and running φ. The planning thread will remain paused until the condition to resume the deliberation is asserted.

Definition 23. (operation for intention deliberation with conflict and goal switching)
Let φ be a desired goal and θ be a ground substitution such that $dom(\theta) = \mathsf{Free}(\varphi)$. Let $\langle run, \varphi_i, \pi_i, \Theta_i\rangle \in \mathcal{P}$.

$$\frac{\langle \varphi, C, redo\rangle \in \mathcal{D},\ \mathcal{B} \models C\theta,\ \mathcal{I} \not\models \varphi\theta,\ \mathcal{P} \not\models \varphi\theta, \\ \exists \varphi_i : \mathcal{I} \models \varphi_i \wedge (\varphi_i \nleftrightarrow \varphi) \wedge (\varphi_i <_p \varphi)}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \cup \{\langle run, \varphi_i, \pi_i, \Theta_i\rangle\}\rangle \rightarrow_\theta \\ \langle \mathcal{B}, \mathcal{D}, \mathcal{I} \cup \{\langle \varphi\theta, C\rangle\}, \mathcal{P}\backslash\{\langle run, \varphi_i, \pi_i, \Theta_i\rangle\} \cup \\ \{\langle run, \varphi\theta, \epsilon, \Theta\rangle, \langle pause, \varphi_i, \pi_i, \Theta_i\rangle\}\rangle}$$

Example 6. (operation for intention deliberation with conflict and goal switching)

Let us consider the following initial configuration.

$$\begin{aligned}
\mathcal{B} =& \{carrying(p), pos(1), conflict(goBase(X), readPamphlet(P)),\\
& priority(goBase(X), \_, 1), priority(readPamphlet(P), \_, 5), \}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
& \langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =& \{\langle readPamphlet(P/p), \{carrying(P/p)\}\rangle\}\\
\mathcal{P} =& \{\langle run, readPamphlet(P/p), \pi_3', \Theta\rangle\}\\
\Gamma =& \{\pi_1 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)\text{''},\\
& \pi_2 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)\text{''},\\
& \pi_3 = \text{“}readPamphlet(P) \leftarrow TRUE \mid Read(P), readPamphlet(P)\text{''}\}\\
\mathcal{A} =& \{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
& \{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\},\\
& \{\}Read(P)\{\}\}
\end{aligned}$$

If, as part of an external action, the belief $base(5)$ *is asserted, the operations in Definition 23 take place, then the situation evolves generating the new status below.*

$$\begin{aligned}
\mathcal{B} =& \{carrying(p), base(5), pos(1), conflict(goBase(X), readPamphlet(P)),\\
& priority(goBase(X), \_, 1), priority(readPamphlet(P), \_, 5), \}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
& \langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =& \{\langle readPamphlet(P/p), \{carrying(P/p)\}\rangle,\\
& \langle goBase(X/5), \{base(X/5), NOT\ pos(X/5)\}\rangle\}\\
\mathcal{P} =& \{\langle pause, readPamphlet(P/p), \pi_3', \Theta'\rangle,\\
& \langle run, goBase(X/5), \epsilon, \{X/5\}\rangle\}\\
\Gamma =& \{\pi_1 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)\text{''},\\
& \pi_2 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)\text{''},\\
& \pi_3 = \text{“}readPamphlet(P) \leftarrow TRUE \mid Read(P), readPamphlet(P)\text{''}\}\\
\mathcal{A} =& \{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
& \{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\},\\
& \{\}Read(P)\{\}\}
\end{aligned}$$

Next, I explain the operations to resume planning threads.

Resuming Intention Deliberation

So far, I have explained that planning threads can either (i) start paused, as a result of an intention deliberation for a conflicting goal with lower priority, or (ii) be paused, as a result of the goal switching process. However, an explanation for how to resume processing is currently missing.

The requirements to resume a paused planning thread are:

1. the intention's conditions still hold with the environment, and;
2. there are no conflicting running planning threads; that is, the conflicting goal has been resolved, dropped, or paused by an external action;
3. the planning thread to be resumed has the highest priority between the paused planning threads;
4. the unification stack is still consistent with the environment or, if not, it can be revised to become consistent.

The first three requirements are intuitive. The last one dictates that in order to resume the execution of a planning thread the deliberation cycle must review the contents of the unification stack for consistency. As the environment could have evolved while the process was paused, the current substitutions must be revised considering the current representation. Let us consider that the implementation provides the meta-operator $\mathsf{revise}(\Theta, \mathcal{B}, \Theta')$ to revise the contents of the unification stack Θ with regards to the current situation $\mathcal{B}$ resulting Θ' with revised values. This operation can be implemented via programming constructs in the supporting logic[8].

The implementation must provide the structures to enforce the execution of this operation. For example, it is possible to associate the event of planning thread completions to the programming constructs to enforce this verification. This solution is adopted for the proof-of-concept implementation presented in Chapter 5.

Definition 24. (operation for resuming planning threads)

Let $\langle \varphi, C \rangle$ *be an intention, and;* $\langle pause, \varphi, \pi, \Theta \rangle \in \mathcal{P}$ *be a paused planning thread for that intention.*

[8] *Variable value replacement*: this operation is implementation-based; for example, if the logic basis is provided by PROLOG, then the values can be updated by variable assignment operations

$$\frac{\mathcal{B} \models C\theta,\ \forall \varphi_i : \mathcal{I} \models \varphi_i \implies \neg(\varphi_i \nleftrightarrow \varphi),\quad \forall \langle pause, \varphi_j, \pi_j, \Theta_j \rangle \in \mathcal{P}, \varphi \neq \varphi_j : (\varphi_j <_p \varphi),\ \mathsf{revise}(\Theta, \mathcal{B}, \Theta')}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \rangle \rightarrow \langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \backslash \{\langle pause, \varphi, \pi, \Theta \rangle\} \cup \{\langle run, \varphi, \epsilon, \Theta' \rangle\} \rangle}$$

In this formula, the first part of the premise verifies whether the intention's conditions still hold with the environment; the second part ensures that the are no conflicting running planning threads; the third part checks whether the planning thread to be resumed has the highest priority between the paused planning threads, and the fourth part runs the meta-operator to verify whether the unification stack is still consistent with the environment. If these conditions are met, then the operation removes the paused planning thread and adds a running one to the revised unification stack to transition the state. I also note that the existing plan π in the paused planning thread is dropped, as the planning thread needs to start to replan for the new conditions.

Example 7. (resuming planning threads)

Consider the following initial configuration. The environment has two bases; the program has a conflict rule that says that the agent cannot try to go to both bases at the same time; and a prioritisation rule that says the agent should go to the closer base first. The intention deliberation process asserts a planning thread to reach $base(1)$ *first – because it is the closest to current position* $pos(3)$ *– and a paused planning thread to reach the destination.*

$$\begin{aligned}
\mathcal{B} =& \{base(10), base(1), pos(3), conflict(goBase(X), goBase(Y)),\\
& priority(goBase(X), \_, distance(pos(Y)))\}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle\}\\
\mathcal{I} =& \{\langle goBase(X/1), \{base(X/1), NOT\ pos(X/1)\}\rangle,\\
& \langle goBase(X/10), \{base(X/10), NOT\ pos(X/10)\}\rangle\}\\
\mathcal{P} =& \{\langle run, goBase(X/1), \pi, \Theta\rangle,\\
& \langle pause, goBase(X/10), \pi', \Theta'\rangle\}\\
\Gamma =& \{\pi_1 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)'',\\
& \pi_2 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)''\}\\
\mathcal{A} =& \{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
& \{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\}\}
\end{aligned}$$

The execution of $goBase(X/1)$ *will lead the agent to the position* $pos(1)$*, in which case the condition* $\{base(X/1), NOT\ pos(X/1)\}$ *is no longer satisfied – i.e. the agent is already in the base – and the intention is retracted. In this configuration, the agent can resume the processing of* $goBase(X/10)$*, however, I highlight that when the planning thread was paused, the agent's position was* $pos(3)$ *and now it is* $pos(1)$*. Hence, the execution of the meta-operator* $\mathsf{revise}(\Theta', \mathcal{B}, \Theta'')$ *will adjust the values to reflect this new condition.*

The deliberation cycle will start to replan for the condition $pos(1)$*. The new configuration is presented below.*

$$\begin{aligned}
\mathcal{B} =& \{base(10), pos(1), conflict(goBase(X), goBase(Y)),\\
& priority(goBase(X), \_, distance(pos(Y)))\}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle\}\\
\mathcal{I} =& \{\langle goBase(X/10), \{base(X/10), NOT\ pos(X/10)\}\rangle\}\\
\mathcal{P} =& \{\langle run, goBase(X/10), \epsilon, \Theta''\rangle\}\\
\Gamma =& \{\pi_1 = \text{``}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)'',\\
& \pi_2 = \text{``}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)''\}\\
\mathcal{A} =& \{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
& \{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\}\}
\end{aligned}$$

Advantages and Limitations

The definitions introduced in this section extend the ones proposed in other work by incorporating a clear representation of conflict resolution and prioritisation rules into the process. For example, 3APL has the "*transition rules for application of condition rules*" as introduced in [Hindriks, 2001, page 43]. However, the support is indirect as it requires the augmentation of the belief base with the representation of conflict rules. The direct representation facilitates the understanding of the process as it displays the features of the deliberation process.

A shortcoming of this model is that it requires the representation of the conflict and prioritisation rules as part of the agent program. Moreover, in order to be effective, it needs to consider all possible combination of running goals and quantify their priorities. A practical solution is to introduce default rules that describe a pre-configured, universal behaviour.

Another shortcoming is that the process of resuming a paused planning thread involves adjusting substitutions in the unification stack prior to the moment when the process can be resumed. As mentioned, it could be implemented in the form of programming constructs that revise the existing substitutions and update the variable with current values.

Finally, one can also claim that the resource utilisation to keep track of several (and potentially large) unification stacks for every running and paused planning thread could compromise the application's feasibility. This situation is especially highlighted in mobile service applications where optimisation of resource utilisation is paramount. Nonetheless, this shortcoming is acceptable given that there are no better solutions. In addition, it is possible to conceive processing optimisations, such as self-monitoring rules that monitor the number of running planning threads and drop those with lower priority where the

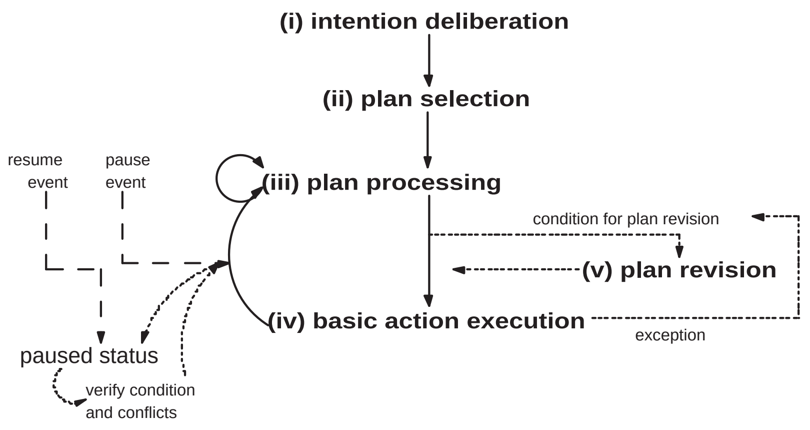

(i) intention deliberation
(ii) plan selection
(iii) plan processing
(iv) basic action execution
(v) plan revision
condition for plan revision
exception
resume event
pause event
paused status
verify condition and conflicts

Figure 3.2: Planning Process

number exceeds a given threshold.

Next, I introduce the planning process.

3.4.2 Planning

Planning is the process of finding solutions to achieve goals. BDI-modelled agents implement *plan on-demand behaviour*, where the planning process infers the next basic actions and sub-goals. Figure 3.2 depicts the operations of the planning process, which are summarised below.

- *(i) intention deliberation* infers the intentions based on desired goals and the current conditions, and adds the planning threads to the plan base. The operations for this function were introduced in the previous sub-section.

- *(ii) plan selection* starts the planning process. In short, after the intention deliberation process (see above), the process selects the plan to follow based on the planning thread's goal.

- *(iii) plan processing* continues to process the elements of the plan base. In short, it resolves the "next step" (i.e. head) of current plans.

- *(iv) basic action execution* processes the basic actions leading the plans of the running planning threads. The execution of basic actions can fail leading to a situation where the plan should be revised.

- *(v) plan revision* promotes the plan revision process if either a condition for plan revision is met or an exception occurs in basic action execution.

In summary, the process works as follows. After the intention deliberation, the planning thread is initialised with a empty plan[9], that is: $\langle run, \varphi, \epsilon, \Theta \rangle$.

In step (ii) the process finds the plan rule $\varphi' \leftarrow C' \mid \pi'$ whose planning thread's goal φ implies the rule's header φ' and the model of the world supports the condition C'. In both cases, there is a set of variable substitutions. Hence, the premise for plan selection is: $\varphi \vDash \varphi'\eta, \mathcal{B} \vDash C'\eta\theta$, where η and θ contains the lists of variable substitutions for the header and condition elements, respectively. However, as there might be multiple plan rules available to resolve the same goal, the process decides which plan to adopt by selecting the one with the lowest cost. I detail these operations in the next sub-sections. This step results in a planning thread configuration like: $\langle run, \varphi, \pi'\eta\theta, \Theta \rangle$.

Next, step (iii) continues to process the plan element. As described in Section 3.3.4, plans are executed by both processing the basic actions at the head of the plan, thus modifying the belief base (step (iii)), and deliberate about the abstract plans at its tail. The process enters in a loop between steps until the plan completes.

Eventually, the process finds a condition for plan revision. In this case, it hooks to the plan revision operation (step (v)). In addition, exceptions in basic action execution trigger the plan revision process in the proposed model.

Moreover, external actions can cause events to pause and resume planning threads. I will exploit this feature in the next section when I introduce the context observer. The next sub-sections detail the operations.

Plan Selection

Plan selection is the process of selecting the plan to follow after the intention deliberation process. In short, the process selects the best plan rule from the planning rules base Γ whose planning thread's goal implies the plan rule's header and the belief base implies the plan rule's condition.

However, as the agent could have been programmed with many alternatives for achieving a goal in a given context, there are potentially many rules to choose from. The deliberation cycle must provide the selection rules used to decide which option to take. Usually, the decision is based on utility maximisation, which means choosing the plan whose basic actions are least costly.

In the BDI-model systems, the use of a theorem prover is assumed to reason off-line about the behaviour of an agent-based system, as explained in Rao [1996]. That is, the reasoned plan rules are loaded into the agent's knowledge bases in a given order. Some systems assume that the rules are

[9]*Intention deliberation*: see Section 3.4.1.

already sorted by least cost to simplify the processing to "*take the first rule*" strategy.

Nonetheless, this strategy does not scale well. Planning rules must be sorted beforehand and provide no flexibility for run-time adjustment or learning. I introduce a function that calculates the costs of selected plans on-demand using the cost rules is a better approach, and I propose to represent this information in the belief base using the representation below.

Definition 25. (cost rules)
The formula $\mathsf{cost}(a, C, n) \in \mathcal{L}$ *represents the cost of executing* $a \in \mathsf{BAct}$ *in the situation* C*, where* n *is a numerical value that quantifies the associated cost; the set of cost formula is defined as* $\mathsf{Cost} \subseteq \mathcal{B}$*.*

Imagine plan $\pi \in \mathsf{Plan}$*, whose "body" part contains the elements* $\pi_b = \{a_1, \ldots, a_n, \pi_s\}$*, with a cost rule for each of the actions in both bodies, so that:* $\forall a_i \in \{a_1, \ldots, a_n\}, \mathsf{Cost} \models \mathsf{cost}(a_i, C_i, n_i)$ *and* $\mathcal{B} \models C_i$*. The function to calculate a plan's cost would be defined as:*

$cost(\pi, C, n)$*, where:*

$$\begin{aligned}
&n = (\textstyle\sum n_i) + cost(\pi_s, C, n),\\
&\forall a_i \in \{a_1, \ldots, a_n\}, \mathsf{Cost} \models \mathsf{cost}(a_i, C, n_i),\\
&\mathcal{B} \models C.
\end{aligned}$$

I highlight that the cost of a plan π is the sum of the costs of the basic actions plus the cost of the abstract plan π_s in the tail of the plan's body. However, in the BDI-model, the planning process behaves following the *think-act* methodology, where the whole plan ahead of execution is not formed. Hence, it is not trivial to apply this recursive formula to calculate the plan cost, because the element $cost(\pi_s, C, n)$ cannot be quantified immediately.

For the sake of conciseness, I introduce the *cost verification operator* $\pi_1 <_C \pi_2$ that reads "π_1 *has a lower cost than* π_2 *in a condition* C", as defined below.

Definition 26. (cost verification operators)

$$\begin{aligned}
&\pi_1 <_C \pi_2 \textit{ iff } cost(\pi_1, C, n_1),\ cost(\pi_2, C, n_2),\ \textit{and } n_1 \leq n_2\\
&\pi_2 <_C \pi_1 \textit{ iff } cost(\pi_1, C, n_1),\ cost(\pi_2, C, n_2),\ \textit{and } n_1 > n_2
\end{aligned}$$

Next, I introduce the plan selection operator. The meta-operator $\mathsf{findprs}(t, P)$ selects the planning rules in Γ whose head match the element t. This operation is defined below.

Definition 27. (find plan rules operator)

$\mathsf{findprs}(t, P)$ *is true if* $P = \{\langle(h_i \leftarrow C_i \mid \pi_i), \eta_i, \theta_i\rangle \mid (h_i \leftarrow C_i \mid \pi_i) \in \Gamma, \mathsf{unify}(t, h_i, \eta_i)$ *and* $\mathcal{B} \models C_i\eta_i\theta_i\}$.

The operation returns the set of tuples $\langle r_i, \eta_i, \theta_i\rangle$, where r_i is the plan rule; the unifier η_i contains the bindings resulting from the unification of the free variables of the plan rule's head with the terms in the goal formula, and; the unifier θ_i contains the bindings produced by the evaluation of the plan rule's guard C_i using the substitutions in η_i from the unification of the head.

For the sake of clarity, I introduce the utility operator *mincost*(.,.) that selects a tuple from the set P, resulting from the operator $findprs(.,.)$, whose plan's cost is the lowest. I highlight that several plans could have the same minimal cost. In this case, the operator selects one plan nondeterministically.

Definition 28. (minimum cost operator)

$\mathsf{mincost}(P, \langle r_i, \eta_i, \theta_i\rangle)$ *if* $\forall\langle r_i, \eta_i, \theta_i\rangle,\ \langle r_j, \eta_j, \theta_j\rangle \in P,\ r_i\eta_i\theta_i <_C r_j\eta_j\theta_j$, *where* $\mathcal{B} \models C$.

Then, the *operation for plan selection* is described below.

Definition 29. (operation for plan selection)
Let $\langle run, \varphi, \epsilon, \Theta\rangle \in \mathcal{P}$ *be a planning thread.*

$$\frac{\mathsf{findprs}(\varphi, P),\ \mathsf{mincost}(P, \langle\varphi_h \leftarrow C \mid \pi_b, \eta, \theta\rangle),\ \mathsf{unify}(\varphi, \varphi_h, \eta),\ \mathcal{B} \models C\theta}{\langle\mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P}\rangle \rightarrow \langle\mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P}\backslash\{\langle run, \varphi, \epsilon, \Theta\rangle\} \cup \{\langle run, \varphi, \pi_b\eta\theta, \Theta'\rangle\}\rangle}$$

The unifier θ contains the bindings produced by the evaluation of the plan rule's guard C' using the substitutions in η from the unification of the head. The evaluation of the guard may compute new bindings θ for free variables in $\varphi_h\eta$. The operation adjusts the unification stack to Θ' by adding the set of substitutions.

Example 8. (plan selection)

Let us consider the agent configuration below, where the intention deliberation operations just asserted the planning thread $\langle run, readPamphlet(p), \epsilon, \Theta\rangle \in \mathcal{P}$

$$
\begin{aligned}
\mathcal{B} =& \{carrying(p), pos(1)\}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
& \langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =& \{\langle readPamphlet(P/p), \{carrying(P/p)\}\rangle\}\\
\mathcal{P} =& \{\langle run, readPamphlet(p), \epsilon, \Theta\rangle\}\\
\Gamma =& \{\pi_1 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)'',\\
& \pi_2 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)'',\\
& \pi_3 = \text{“}readPamphlet(P) \leftarrow TRUE \mid Read(P), readPamphlet(P)''\}\\
\mathcal{A} =& \{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
& \{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\},\\
& \{\}Read(P)\{\}\}
\end{aligned}
$$

As part of the first step in the plan deliberation process, the operation selects the plan rule from Γ *that is able to resolve the goal* $readPamphlet(p)$*. In this case, the plan rule* $readPamphlet(P/p) \leftarrow TRUE \mid Read(P/p), readPamphlet(P/p)$ *is selected. The next agent configuration becomes:*

$$
\begin{aligned}
\mathcal{B} =& \{carrying(p), pos(1), base(5)\}\\
\mathcal{D} =& \{\langle goBase(X), \{base(X), NOT\ pos(X)\}, true\rangle,\\
& \langle readPamphlet(P), \{carrying(P)\}, false\rangle\}\\
\mathcal{I} =& \{\langle readPamphlet(P/p), \{carrying(P/p)\}\rangle,\\
& \langle goBase(X/5), \{base(X/5), NOT\ pos(X/5)\}\rangle\\
\mathcal{P} =& \{\langle run, readPamphlet(P/p), (Read(p), readPamphlet(p)), \Theta'\rangle,\\
& \langle run, goBase(X/5), \epsilon, \{X/5\}\rangle\}\\
\Gamma =& \{\pi_1 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X < Y \mid GoRight(), goBase(X)'',\\
& \pi_2 = \text{“}goBase(X) \leftarrow pos(Y)\ \&\ X > Y \mid GoLeft(), goBase(X)'',\\
& \pi_3 = \text{“}readPamphlet(P) \leftarrow TRUE \mid Read(P), readPamphlet(P)''\}\\
\mathcal{A} =& \{\{pos(X)\}GoRight()\{NOT\ pos(X), pos(X+1)\},\\
& \{pos(X)\}GoLeft()\{NOT\ pos(X), pos(X-1)\},\\
& \{\}Read(P)\{\}\}
\end{aligned}
$$

Plan Processing

The function for *plan processing* continues the execution of the plan. In short, it resolves the “next step” (i.e. head) of current plans. As stated in Section 3.3.4, the term t_h in the plan's head can be either $t_h \in \mathsf{AbstractPlan}$ or $t_h \in \mathsf{BAction}$. Let us assume the first situation.

Definition 30. (operation for plan execution)

Let $\langle run, \varphi, t_h; \pi, \Theta\rangle \in \mathcal{P}$ *be a planning thread.*

$$\frac{t_h \in \mathsf{AbstractPlan},\ \mathsf{findprs}(t_h, P),\ \mathsf{mincost}(P, \langle \pi_h \leftarrow C \mid \pi_b, \eta, \theta \rangle), \\ \mathsf{unify}(t_h, \pi_h, \eta),\ \mathcal{B} \models C\eta\theta}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \rangle \rightarrow \langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \backslash \{\langle run, \varphi, t_h; \pi, \Theta \rangle\} \cup \{\langle run, \varphi, (\pi_b; \pi)\eta\theta, \Theta' \rangle\}\rangle}$$

Similarly, the unifier θ contains the bindings produced by the evaluation of the plan rule's guard C using the substitutions in η from the unification of the head. The evaluation of the guard may compute new bindings θ for free variables in $\pi_h\eta$. The substitutions apply to both parts of the plan, so $(\pi_b; \pi)\eta\theta$. The operation adjusts the unification stack to Θ' by adding the set of substitutions.

Basic Action Execution

Continuing the explanation above, let us consider the second situation where $t_h \in \mathsf{BAction}$.

I am using the definition of basic action execution from 3APL, presented in Hindriks [2001]. That is, basic actions update the belief base through a the partial function $\mathcal{T}$, where $\mathcal{T}(a, \mathcal{B})$ returns the result of updating the belief base $\mathcal{B}'$ by performing the action a. The fact that $\mathcal{T}$ is a partial function represents that the action may not be executable in some belief states. After execution of an action, the action is removed from the plan. The operation is described below.

Definition 31. (operation for basic action execution)
Let $t_h \in \mathsf{BAction}$ *and* $\langle run, \varphi, (t_h; \pi), \Theta \rangle \in \mathcal{P}$ *be a planning thread.*

$$\frac{t_h \in \mathsf{BAction},\ \mathcal{T}(t_h, \mathcal{B}) = \mathcal{B}'}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \cup \{\langle run, \varphi, (t_h; \pi), \Theta \rangle\}\rangle \rightarrow \langle \mathcal{B}', \mathcal{D}, \mathcal{I}, \mathcal{P} \cup \{\langle run, \varphi, \pi, \Theta \rangle\}\rangle}$$

The problem of unexpected events in dynamic environments cannot be overlooked. It is possible that basic actions fail to execute due to instabilities or unexpected situations. One expects autonomous applications to be prepared to handle these situations and maintain coherence and consistency. In order to achieve this ideal the application must provide an exception recover strategy. Moreover, the system can detect the error condition either via an external logic that compares expected with actual results, or programming constructs that capture exceptions from an actuation's execution. I propose a practical solution in Chapter 5.

Plan Revision

Plan revision is a required component to support the deliberative behaviour of BDI-model agents. This facility helps to provide stability to the deliberation process. It is useful both for recovering from a bad plan that leads to undesirable states, and an exception in basic action execution. In both cases, the plan revision process is part of the recovery strategy.

In this work, I present a practical approach for the plan revision process. I am not detailing the semantics of the operations; rather referring to the extensive discussion introduced in van Riemsdijk [2006].

Two factors can trigger the processing: a favourable condition for plan revision, which is implied from the guards of the plan revision rules, or exceptions when executing basic actions, which requires the revision of the current plan as part of the plan recovery strategy. The belief base is not changed through plan revision. The operation is defined below.

Definition 32. (operation for plan revision)
Let $\pi_h \leftarrow C \mid \pi_b \in PR$ be a plan revision rule, and $\langle run, \varphi, \pi; \pi', \Theta\rangle \in \mathcal{P}$ be a planning thread, such that $\pi = \pi_h\theta$.

$$\frac{\pi_h \leftarrow C \mid \pi_b \in PR,\ \pi = \pi_h\theta,\ \mathcal{B} \models C}{\begin{array}{c}\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P}\rangle \rightarrow \\ \langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P}\backslash\{\langle run, \varphi, \pi; \pi', \Theta\rangle\} \cup \{\langle run, \varphi, (\pi_b; \pi')\theta, \Theta'\rangle\}\rangle\end{array}}$$

Essentially if there is a plan revision rule whose head π_h is implied by the head of the current plan $\pi; \pi'$, resulting the set of substitutions in θ. Then, the body of the plan revision rule replaces the head of the current plan, and; the substitutions in θ apply to the new plan. The unification stack is adjusted to Θ' by adding the substitutions in θ. In addition, as the revision operation only affects the processing of one thread, the operation of parallel threads continues untouched; providing for very general and flexible plan revision capabilities.

Consequences of Plan Revision. The first question related to plan revision asks *when to revise running plans*? So far, I have suggested reactive strategies to respond to exceptional situations in the plan execution process, however, a better balance between reactive and proactive behaviour is reached if the plan revision process considers the possible future situations the current course of actions might lead to. That is, besides *reactive plan revision* it is also possible to have *proactive plan revision* based on current execution context.

Rao [1996] proposes a dilemma regarding the frequency of the plan revision process: "*reconsidering the choice of action at each step is potentially too expensive and the chosen action possibly invalid, whereas unconditional commitment to the chosen course of action can result in the system failing to achieve its objectives.*" It proposes that it is possible to limit the frequency of reconsideration and thus achieve an appropriate balance between too much reconsideration and not enough; assuming that potentially significant changes can be determined instantaneously. Schut et al. [2004] also analyses the issue and presents a model for BDI agents that can determine their own intention reconsideration strategy based on future goals and arising opportunities.

Notice that *how* and *when* plans are revised are, ultimately, application specific issues. The implementation must provide the structures to enforce execution of the plan revision rules. For example, the proof-of-concept implementation described in Chapter 5 provides a practical solution where the function to enforce plan revision is called after the plan selection operations execute.

Advantages and Limitations

Thus far, I have detailed the operations of the agent's deliberation, and also provided semantics to deliberate intentions and to compose plans from the practical reasoning rules; presented as the intention deliberation and plan formation processes, respectively. Transition functions that process the basic actions execute composed plans. Several plan formation processes run in parallel, in structures called planning threads, which the deliberation cycle is able to pause, resume or stop at anytime, minding the requirements for coherence, conflict resolution, and prioritisation. This feature enables concurrent processing; a fundamental requirement of applications that execute in highly dynamic environments.

This model offers advantages for: (i) creating more stable deliberation cycles, equipping them to handle unexpected events, and (ii) applying this information to self-adjust.

The concept of on-demand plan selection based on cost calculation provides a clear picture of the operations and how they can be optimised. However, in order to work, this method requires representing the costs for all conceivable basic actions. As a practical solution, it is possible to create *generic cost rules*, e.g. considering broad conditions, which offers an opportunity to *adapt* the processing, through either the learning or adjusting costs.

The shortcoming is that it is not possible to assure the *best cost*. Incomplete information may lead the planner system to take a path with an immediate lower cost, but which ultimately adds up to a higher cost as the situation progresses. In other words, this plan selection process results in minimum locals derived from sub-goals and conditions. This is a limitation of the plan on-demand behaviour, adopted by the BDI-model.

As discussed in previous sub-section, the plan revision process is an essential component in the BDI-model deliberation cycle. It allows the application to recover from unexpected situations, introduces a degree of application stability, and supports the *exception recover strategy*, which improves the application's performance.

However, the plan revision strategy is only as good as the quality of the plan revision rules. Although it is possible to conceive generic plan revision rules, the quality of the plan revision process is greatly influenced by the application's know-how, programming skills, and existing model of the world.

One drawback, this approach requires the programmer to specify the rules for every foreseeable exception *a priori*. This would be developmentally demanding, error prone, and against a principle of autonomy in agent computing, where an agent should be situated and able to handle the unexpected. In fact, if it was possible for the agent to know all possible conditions and react to the situations accordingly, then this would inherently be a reactive agent.

I propose that the deliberation cycle must include the structures to represent the semantics of exception handling. Sophisticated strategies could be composed to provide higher levels of stability to the deliberation cycle; however, I shall not enter in the semantic issues of exception manipulation as they are outside the scope of this thesis. I give a practical solution, based on the representation of exceptions as internal events and the use of the coordination structure to handle these events in Chapter 5.

What Comes Next? The application can use environmental information – i.e. events on the window of opportunity – to help improve the application performance. This feature would contribute to the shortcoming of current approaches, which lack proper integration of context information to enhance the process. That is, current approaches are either *too reactive*, such as context-awareness based solutions that tend to interrupt the processing when certain situations arise, without considering the stability (i.e. coherence) of the goal-oriented application –, or *too proactive*, such as the deliberation cycle of some of current agent based solutions that are overcommitted to the planning phase. Thus, no matter what the situation or how urgent the need for action, these systems always spend more time to plan and reason about achieving a given goal before sensing the environment and performing any external actions.

In what comes next, I propose an extended model that exploits environmental information for the reconsideration decision. This model aims to improve the balance between reactiveness and proactiveness when the application is operating in highly dynamic environment.

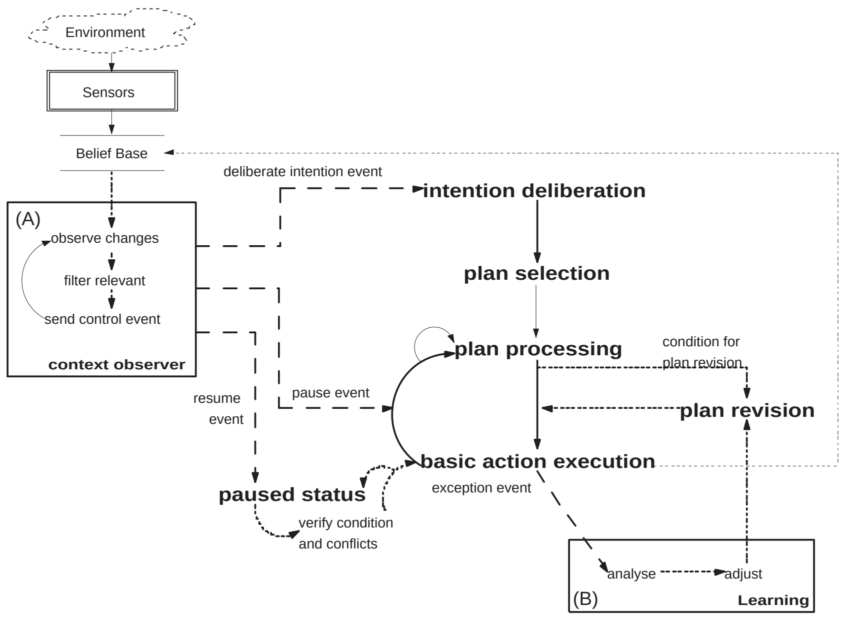


Figure 3.3: Conceptual Model

3.5 Conceptual Model

In order to design a system capable of reviewing the current processing in reaction to changes of the environment, one must implement a process between (or within) the plan selection rule and basic action execution steps that reconsiders the plan. It is intuitive that the computation involved in monitoring the operational conditions is proportional to the number of updates in the representation of the world.

Figure 3.3 depicts the conceptual model. I propose an design where an external module exploits the structures to pause and resume the planning threads. Moreover, exceptions in basic action execution are captured to trigger the adaptation and recovery strategy. In what follows, I integrate these structures with those specialised in (i) observing and making sense of changes of the environment, i.e. the *context observer*, and (ii) analysing and adjusting the operational parameters as part of the recovery strategy, i.e. the *learning module*. I detail the elements and the intra-module coordination in the next sub-sections.

3.5.1 Context Observer

It is intuitive that the computation involved in monitoring the conditions for operations described in the previous sections, and the *reconsideration* of actions, is proportional to the number of updates in the representation of the world – the so-called *degree of dynamism.* One can argue that, to respond in a timely manner, the more dynamic the environment, the more reactive a real-time information system must be! Georgeff et al. [1987] proposes that:

> "*The real-time constraints imposed by dynamic environments require that a situated system be able to react quickly to environmental changes. This means that the system should be able to notice critical changes in the environment within an appropriately small interval of time. However, most embedded planning systems provide no mechanisms for reacting in a timely manner to new situations or goals during plan execution, let alone during plan formation*".

I highlight the relationship between these definitions and the concepts of *plan ahead behaviour* and *plan on-demand behaviour.* Both approaches have advantages and disadvantages. Plan ahead behaviour is unsuited for dynamic environments, where changes of the environment can invalidate a previously determined sequence of steps; thus their execution results in *unexpected situations.* Plan on-demand behaviour offers the advantage of implementing few, immediate actions upon the (visible, thus predictable) environment, before planning for continuation *on-demand*, however its execution can be resource demanding. This behaviour is the focus of current implementations and the basis of the BDI-model deliberative behaviour.

For example, in Rao and Georgeff [1995]'s BDI-model, the changes are captured by *get-new-external-events()* interactions in the deliberation cycle. The process is described as: "*(...) any external events that have occurred during the interpreter cycle are then added to the event queue. (...) At the beginning of every cycle, the option generator reads the event queue and returns a list of options.*"

Nonetheless, this approach is not practical in agents that operate in highly dynamic environment, due to the amount and frequency of context information updates. The *loop structure* introduces a delay between the moment when the observation is made and its processing. In addition, it is not efficient in concurrent planning, due to redundant calls to *get-new-external-events()* method. The work in Schut et al. [2004] discusses variations of these behaviours where the reconsideration happens between the bold and cautious behaviours.

I propose to extend the deliberation cycle with a structure specialised in observing the environment: the *context observer module.* It works by observing changes in the environment, rationalising the changes that are relevant to the current processing, and sending control events to the planning threads. It is assumed that the application has a sensor that collects information from the environment and updates the belief base, and that updates to any of the knowledge bases – i.e. belief, desire, intention, and plan bases – generate events that can be captured and manipulated by the application.

Similar functionality has been suggested in other agent platforms, for example, 3APL has the *interruptions* and *disruptors*, as described in [Hindriks, 2001]; 2APL has *environment events*, as described in [Dastani et al., 2007], and; AgentSpeak has *observables*, as described in [Inverno and Luck, 1998]. However, the proposed design provides a clear view of the programming structures required to support the context observation functionality.

On the other hand, Zhang and Huang [2006] proposes a framework where parallel BDI agents may be built, with different configurations depending on the availability of physical resources. That work claims that agent applications built atop this framework offer a number of advantages over the sequential BDI-model, *viz.*: (i) enhanced reactiveness, where changes in the agent's environment can be detected immediately, and (ii) impromptu reconsideration of desires and intentions. However, the framework focuses on the issue of concurrent processing in BDI-model and does not provide a clear method to infer *when* to revise the current processing in reaction to environmental events.

In this sub-section, I propose a structure based on a central observer and an intra-module coordination architecture. The architecture, depicted in Figure 3.3(A), works as follows:

1. the module observes changes in the environment;
2. the changes that are relevant for current processing are filtered in, and;
3. the module sends events signalising the occurrences to the planning threads.

The implementation must provide the structures to observe changes in the environment as a practical solution. For example, the proof-of-concept implementation in Chapter 5 adopts a practical solution where events related to updating the belief base trigger the context observer processing.

Moreover, it is intuitive that because the belief base can be large and several updates can occur simultaneously in a highly dynamic environment, the application should have the means to rationalise the processing of incoming events. The best solution is to filter what is relevant and discard the rest.

Let us consider that β is an update of the belief base configuration due to the execution of a basic action; that is, if the execution of the action a upon the belief base $\mathcal{B}$ results in the new configuration $\mathcal{B}'$, then $\beta = \mathcal{B}' \backslash \mathcal{B}$. I introduce the function $\mathsf{impact}(\beta, \varphi, V_r)$ that assesses the impact of the belief β on the processing of goal φ, and V_r is a free variable that receives the result, which can be either of the configurations:

- $\mathsf{impact}(\beta, \varphi, false)$ if the belief β has no impact on the processing of φ.
- $\mathsf{impact}(\beta, \varphi, start)$ if the belief β implies starting the processing of φ.
- $\mathsf{impact}(\beta, \varphi, pause)$ if the belief β implies pausing the processing of φ.
- $\mathsf{impact}(\beta, \varphi, resume)$ if the the belief β implies resuming the processing of φ.

This function helps to rationalise the computation by estimating the impact of a belief base update on the current configuration. I propose to implement this functionality as a programming construct in the application. The next chapter provides further detail on how to operationalise this structure.

Next, I revise the operations introduced in Section 3.4 and describe how events from the context observer impact the operations of the deliberation cycle.

3.5.2 Revised Operations

I propose that the function to rationalise the processing of incoming events helps to rationalise the computation and must be considered as part of the premise in the operations for intention deliberation, pausing and resuming planning threads. This combination can be seen as a logical “hook up” between the context observer module with the operations of the deliberation cycle. Besides helping in the computational performance, it also provides a solution for the problem of *when* to pause and resume deliberations, as I shall exploit in the next chapter.

Let us consider the operations for intention deliberation introduce in Section 3.4.1. The operation for intention deliberation with no conflicts, introduced in Definition 21, can be revised as follows.

Definition 33. (revised operation for intention deliberation with no conflicts) *Let $\langle \varphi, C, redo \rangle$ be a desire; β be an update in the belief base configuration, and; θ be a ground substitution such that $dom(\theta) = \mathsf{Free}(\varphi)$.*

$$\frac{\begin{array}{c}\langle \varphi, C, redo \rangle \in \mathcal{D}, \mathsf{impact}(\beta, \varphi, start),\ \mathcal{B} \models C\theta,\ \mathcal{I} \not\models \varphi\theta, \\ \nexists \varphi_i : \mathcal{I} \models \varphi_i \wedge (\varphi \not\leftrightarrow \varphi_i)\end{array}}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \rangle \rightarrow_\theta \langle \mathcal{B}, \mathcal{D}, \mathcal{I} \cup \{\langle \varphi\theta, C \rangle\}, \mathcal{P} \cup \{\langle run, \varphi\theta, \epsilon, \Theta \rangle\} \rangle}$$

Similarly, the operation for intention deliberation with conflicts and lower priority, introduced in Definition 22, can be revised as below.

Definition 34. (revised operation for intention deliberation with conflict and lower priority)
Let $\langle \varphi, C, redo \rangle$ be a desire; β be an update in the belief base configuration, and; θ be a ground substitution such that $dom(\theta) = \mathsf{Free}(\varphi)$.

$$\frac{\langle \varphi, C, redo \rangle \in \mathcal{D}, \mathsf{impact}(\beta, \varphi, start),\ \mathcal{B} \vDash C\theta,\ \mathcal{I} \nvDash \varphi\theta,\ \mathcal{P} \nvDash \varphi\theta, \\ \exists \varphi_i : \mathcal{I} \vDash \varphi_i \land (\varphi_i \nleftrightarrow \varphi) \land (\varphi <_p \varphi_i)}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \rangle \rightarrow_\theta \langle \mathcal{B}, \mathcal{D}, \mathcal{I} \cup \{\langle \varphi\theta, C \rangle\}, \mathcal{P} \cup \{\langle pause, \varphi\theta, \epsilon, \Theta \rangle\} \rangle}$$

Moreover, the operation for intention deliberation with conflict and goal switching, introduced in Definition 23, is revised below.

Definition 35. (revised operation for intention deliberation with conflict and goal switching)
Let $\langle \varphi, C, redo \rangle$ be a desire; β be an update in the belief base configuration, and; θ be a ground substitution such that $dom(\theta) = \mathsf{Free}(\varphi)$.

$$\frac{\langle \varphi, C, redo \rangle \in \mathcal{D},\ \mathsf{impact}(\beta, \varphi, start),\ \mathcal{B} \vDash C\theta,\ \mathcal{I} \nvDash \varphi\theta, \\ \mathcal{P} \nvDash \varphi\theta, \exists \varphi_i : \mathcal{I} \vDash \varphi_i \land (\varphi_i \nleftrightarrow \varphi) \land (\varphi_i <_p \varphi)}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \cup \{\langle run, \varphi_i, \pi_i, \Theta_i \rangle\} \rangle \rightarrow_\theta \\ \langle \mathcal{B}, \mathcal{D}, \mathcal{I} \cup \{\langle \varphi\theta, C \rangle\}, \mathcal{P} \backslash \{\langle run, \varphi_i, \pi_i, \Theta_i \rangle\} \cup \{\langle run, \varphi\theta, \epsilon, \Theta \rangle, \langle pause, \varphi_i, \pi_i, \Theta_i \rangle\} \rangle}$$

In these three revised operations, the premise offers a computational optimisation over the original definition with the presence of the operation $\mathsf{impact}(\beta, \gamma, start)$ to verify the condition while considering the environment status. Now, only relevant belief base updates initiate verifying $\mathcal{B} \vDash C\theta$, which is an expensive operation in computational terms. I highlight that in the original definition any modification would trigger verification of that condition.

Pausing Executions

In Section 3.4.1, I mentioned that planning threads can be paused by an external action, however, I have not yet described the conditions that trigger this action. The $\mathsf{impact}(\beta, \varphi, V_r)$ provides a solution for this issue. It evaluates as $V_r = pause$ if the belief β implies pausing the processing of φ. Hence, the operation to pause a planning thread is defined below.

Definition 36. (operation for pausing planning threads)
Let $\langle \varphi, C \rangle \in \mathcal{I}$ be an intention and $\langle run, \varphi, \pi, \Theta \rangle \in \mathcal{P}$ the planning thread related to that goal. Let β be an update in the belief base configuration.

$$\frac{\mathsf{impact}(\beta, \varphi, pause)}{\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \rangle \rightarrow \langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \backslash \{\langle run, \varphi, \pi, \Theta \rangle\} \cup \{\langle pause, \varphi, \pi, \Theta \rangle\} \rangle}$$

That is, if the belief β implies pausing the processing of φ, then the status of the planning thread is updated to *pause*. This rule enables operating the context observer as a component. The conditions that result in $\mathsf{impact}(\beta, \varphi, pause)$ are related to events of the window of opportunity, which is discussed in the next chapter.

Resuming Executions

Similarly, I introduce the operation to resume planning threads in Section 3.4.1. As mentioned, external events, such as planning thread completions, could trigger resuming planning threads. Here, I also suggest that events associated with the windows of opportunity can be exploited to trigger the resume operation. The operation defined below takes this condition into account and extends the functionality for resuming execution.

Definition 37. (extended operation for resuming planning threads)

Let $\langle \varphi, C \rangle$ be an intention, and; $\langle pause, \varphi, \pi, \Theta \rangle \in \mathcal{P}$ be a paused planning thread for that intention.

$$\frac{\begin{array}{c}\mathsf{impact}(\beta, \varphi, resume), \mathcal{B} \vDash C\theta,\ \forall \varphi_i : \mathcal{I} \vDash \varphi_i \wedge \neg(\varphi_i \nleftrightarrow \gamma),\\ \forall \langle pause, \varphi_j, \pi_j, \Theta_j \rangle \in \mathcal{P}, \varphi \neq \varphi_j : (\varphi_j <_p \varphi),\ \mathsf{revise}(\Theta, \mathcal{B}, \Theta')\end{array}}{\begin{array}{c}\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \rangle \rightarrow\\ \langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P} \backslash \{\langle pause, \varphi, \pi, \Theta \rangle\} \cup \{\langle run, \varphi, \epsilon, \Theta' \rangle\} \rangle\end{array}}$$

The premise in this formula extends the Definition 24 by adding the $\mathsf{impact}(\beta, \varphi, resume)$, which can be described as an event generated by the context observer in response to events associated with the window of opportunity. This rule offers a computation optimisation, as it enables operation of the context observer to evaluate the impact of environmental changes upon the intention execution, as discussed in the next chapter.

Advantages and Limitations

The context observer helps to improve the deliberation performance by filtering relevant events and promoting real-time coordination between observation and plan execution. This approach makes use of environmental information to support the decision on *when* to reconsider, while the planning threads offer a solution on *how* to reconsider. I propose that being able to pause and resume the current processing threads saves the application resources, and causes faster reaction times when the agent is back within the window of opportunity. A summary of the achieved benefits is presented below:

- *increases reaction time*: the relevance filters cause events to be sent only when a relevant change happens, while the planning threads allow saving previously executed deliberation;
- *saves processing time*: both on belief base operations, intention deliberation, and plan execution;
- *acts coherently*: avoiding executing action at inopportune moments, as the execution is paused when the agent moves out of the window of opportunity;
- *acts consistently*: the deliberation cycle provides a BDI-model deliberation system, which works in a goal-oriented fashion.

I present case studies demonstrating the performance improvement in Chapter 6.

3.5.3 Learning

In Section 3.2.4, it was mentioned that agents must be able to adapt their deliberative behaviour to improve performance. As mentioned in Section 3.2.4, there are several methods for implementing learning and adaptation. The direct and simple approach is to learn in reaction to environmental exceptions; while more complex approaches learn based on indirect facts and introspection.

For the sake of simplicity, I propose a view where adaptation actions occur either in response to exceptions, generated by unexpected events during plan execution, or as part of the conflict resolution process. I call the process of analysing and adjusting the execution parameters in response to a failure to execute a basic action (i.e. an exception event) as “reactive learning”.

I also focus on utility learning, which is learning how to adjust the agent’s configurable parameters and agent’s knowledge in order to positively influence the processing of current intentions.

Essentially, changes can occur in two ways: either *self-modification* as an adaptive change to the deliberation cycle, or *learning* by accumulating knowledge. It assumes that the agent provides access to the structures that compose the deliberation process, that is, to the metareasoner. This structure contains the set of meta-reasoning information and the elements to reason about *how* to reason.

Schmill et al. [2007] identifies four primary qualifications required to implement the learning process:

- (i) need for information: the metareasoner need access to enough information about what its host is doing so that it can detect when expectations about the host's behaviour are being violated;
- (ii) ability to make recommendations: metareasoner is able to make recommendations of targeted changes to the host system to address expectation violations;
- (iii) ability to monitor performance improvement: the metareasoner is able to use the first two capabilities to monitor how well a recommended change has addressed the problem, and;
- (iv) real-time learning.

In this work, I do not explore details on how to learn new knowledge, as that is outside my scope. Techniques for machine learning, presented in Chipman and Meyrowitz [1993], can be applied for that objective. For those interested, inspiration can also be taken from the SOAR project that applies learning to adjust its rules, *viz* Lehman et al. [2006].

How to learn?

Doyle and Thomason [1999] proposes that, when confronting a problem, the decision maker "*identifies alternatives, outcomes, probabilities, and utilities through an iterative process of hypothesising, testing, and refining a sequence of tentative formulations.*" The agent identifies the alternatives and outcomes through direct queries or knowledge of the circumstances that lead to the exception. Thus the actions of *reviewing* the belief base entries, *revising* the plan, and *adapting* the rules of operation are the core of the so-called *adaptive behaviour*.

I stress that, in this work, I am not detailing learning operations. The learning module must implement the functions to analyse and adjust meta-reasoning information, using the provided meta-operators; operations activated by events originating from run-time exceptions, such as failures to execute basic actions. In addition, the internal learning functions must coordinate the plan revision of the current processing. These operations are part of the recovery strategy, introduced in Section 3.4.2.

There are a number of possible run-time adjustments throughout this chapter, as summarised below:

1. *adjust conflict rules*: if the exception is fired because two goals conflict on the use of a same resource, then it is possible to adjust the conflict rules to avoid these goals processing in parallel; adding a new conflict rule to the belief base as the term $\mathsf{conflict}(t_1, t_2)$, *viz.* Definition 16;

2. *adjust prioritisation rules*: when two conflicting rules are to be processed in parallel, the deliberation cycle prioritises the one that is marked as more urgent. The prioritisation rules exist as terms in the belief base in the form $\mathsf{priority}(\varphi, C, n)$, *viz.* Definition 19; the adjusting of priority rules that influence the intention processing;

3. *adjust cost rules*: when multiple plan rules are selected in a given situation, the deliberation cycle selects the one whose basic actions have the lower execution cost. The cost rules exist as terms in the belief base in the form $\mathsf{cost}(a, C, n)$, *viz.* Definition 25; with each rule representing the cost of executing the basic action a in a given condition. Adjusting cost rules influences the behaviour of the plan selection function.

On the other hand, it is possible to realise an *external learning* structure that recommends new plan rules and plan revision rules be added to the agent's knowledge base, by means of communication or direct access to agent's structures. This new information is added to the knowledge base and applied by the deliberation process, hopefully positively.

Moreover, in restricted domains, the agent might have *self-adjustment rules* preset in the knowledge databases, which are used as part of the deliberation process to learn new rules of operation. For example, consider negotiation agents that could adjust their rules of operation about the negotiation activity. Because the domain is very specific, it is possible to conceptualise a number of *self-adjustment* and *self-monitoring* rules *a priori*. In some situations, the former would adjust the program behaviour and the latter observes the new performance to approve or reject the modification.

Finally, there are several other possibilities in the field of *machine learning*. For example, Maes [1994] proposed an alternative approach for the knowledge-based solution, where the agent is preset with extensive domain-specific background knowledge:

> "*(...) an alternative approach to building interface agents that relies on machine learning techniques. The hypothesis that is tested*

> *is that, under certain conditions, an interface agent can program itself, i.e. it can acquire the knowledge it needs to assist its user. The agent is given a minimum of background knowledge, and it learns appropriate behaviour from the user and from other agents. The particular conditions that have to be fulfilled are: (1) the use of the application has to involve a substantial amount of repetitive behaviour, and (2) this repetitive behaviour is potentially different for different users.*"

As the topic of learning is out of the scope of this work, I consider it sufficient to provide a concept of the structures required for its operations.

3.6 Conclusion

In this chapter, I proposed an extended computational model for BDI inference systems that takes advantage of environmental events to coordinate the deliberation process. It allows the deliberation process to be adjusted in response to impromptu events, however this structure alone is not enough for dealing with environmental changes. I introduce the context observer to decide when the deliberation process should be adjusted; using the windows of opportunity to decide which context information is relevant, thus rationalising the processing.

I also introduced a syntax for the deliberation process and described the internal operations. The presentation provides a clear picture on *why* agent applications must be able to adapt, *what actions* are implied by adaptation, and,*what structures* are required to support these actions.

Further, I introduced a conceptual model that integrates the elements to observe changes of the environment and signals the deliberation cycle on how to react to these events. In summary, the model is composed of:

1. *context observer module* observes changes of the environment and rationalises relevant events that signal the planning module when to reconsider an intention's processing thread;

2. *planning module* implements the operations for intentions' planning threads, including plan selection, plan revision, and basic action execution, and;

3. *learning module* promotes utility learning, by proving the structures to plug-in external functionality to adjust the configurable parameters and knowledge.

I conclude that the proposed model supports the requirements to (i) support application's interactions, facilitating the application's design process; (ii) provide quality information by implementing a flexible, adaptive inference system capable of adjusting the processing line in response to internal and external conditions, and (iii) optimising the processing performance, thus reducing resource utilisation.

As a limitation, the learning process works by reacting to exceptions in the executions. That is, the process of analysing and adjusting the execution parameters is triggered by exception events, resulting from a failure to execute a basic action. Therefore, the proposed structure produces reactive learning, after the problem has happened. An optimised structure could also implement pro-active learning, by observing the environmental and processing conditions; however this topic (i.e. learning) is out of the scope of this work.

In the next chapter, I describe how to integrate these definitions to structures that reason on predictable time-space situations in mobile environments.

Chapter 4

Design Model

"Design is directed toward human beings. To design is to solve human problems by identifying them and executing the best solution."
Ivan Chermayeff.

4.1 Introduction

The previous chapter introduced the conceptual model for an enhanced inference system equipped to cope with dynamic environments and able to reason about multiple, conflicting options, which comprised a *context observer* to rationalise relevant changes of the environment in order to optimise the application's performance, a *planning module*, and a *learning module* to promote utility learning.

In this chapter, I extend these definitions and introduce the *design model*. The explanation provides practical solutions to pending issues of the conceptual model, such as the elements to take advantage of environmental information. In addition, the application is equipped to exploit events associated with the window of opportunity to optimise the processing.

The chapter is organised as follows. Section 4.2 introduces the motivation for the design model. Section 4.3 details the elements and interactions, while Section 4.3.4 puts the pieces together and explains how the process works. Finally, the chapter concludes with the fifth Section 4.4.

4.2 Motivation

The main argument throughout this work is that mobile services operate in an information rich environment where the application must be equipped to process more information using fewer resources. Therefore, the key design issue

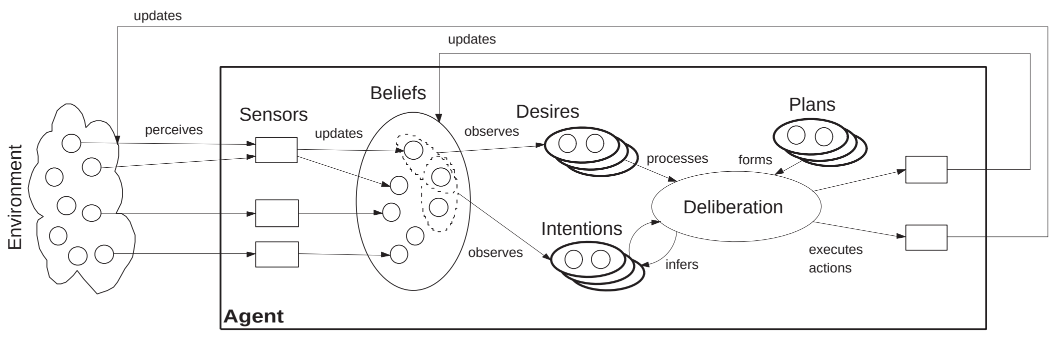


Figure 4.1: Perception and Relevance in Agent Systems

relates to *optimisation* in order to improve performance. Kinny and Georgeff [1991] proposed that:

> "*The crucial problem facing designers of situated agents (...) is to ensure that the agent's response to important changes in the environment are both appropriately and timely. These requirements appear to conflict, since the reasoning seems to be needed to choose appropriate actions could require arbitrarily large amount of time to perform.*"

This issue is highlighted in mobile service environments. Opportunities exist to combine context awareness to an extended BDI-modelled deliberation cycle in order to improve an application's performance. This way, the application would be equipped to exploit events associated with environmental behaviour to coordinate current processing threads. In other words, the process uses this information to support the decision on *when* to reconsider!

To support this development, I introduce the *context observer* module that works as a central observer for changes in the environment. This structure helps to improve the application's performance by filtering relevant events, and allows real-time coordination between observation and plan execution.

The requirements for this development are impromptu reconsideration of in-processing elements, taking advantage of events from the window of opportunity, maintaining application coherence amidst environmental instabilities, and executing with resource constraints. In this chapter, I describe how to integrate the application's elements to support these requirements.

Next, I introduce some background information on the related concepts.

Perception, Relevance and Opportunity

This study aims to explore how to make use of events associated with context changes to optimise the processing. Firstly, I want to examine which updates are relevant, to enable defining the operations for the filters of relevant information; secondly, I analyse how this information can be used to optimise the application's behaviour.

Figure 4.1 provides the conceptual view of the agent's structures. *Sensors* update a subset of the belief base with information collected from the environment. Belief base entries are also updated through introspection. However, only a subset of the updated information is actually relevant to the current processing.

This concept is intuitive as the agent must verify the commitment and termination conditions associated with desires and intentions in order to operationalise the deliberation. The agent can rationalise the process of perceiving the world if its inference system is equipped to classify and select the relevant changes and filter only the ones that are important to current processing. The leading question is: *how to exploit features of the dynamic environment to improve the performance of the application?*

As a solution, the application can recognise opportunities for optimising its performance by being aware of the various contexts it resides in and realising when changes in these contexts are relevant for the current processing. To that end, it is possible to describe *proximity conditions* of a goal. One can think of these conditions as events relating to the window of opportunity where the information is convenient. Several dimension can be applied, such as the segments of context information, proposed in Graham and Kjeldskov [2003]: time, absolute location, relative location, objects present, activity, social setting, environment, and culture.

It is possible to think in terms of being *in* or *out* of the window of opportunity and formulate yes-no questions to reflect these conditions. For example: are the objects present? Is the user engaged in certain activities? Is the user in a certain social setting? Are the conditions valid? These are the *proximity conditions* of a goal.

Hence, if the application is equipped with a flexible deliberation cycle, then it can adapt its processing threads in response to changes of these conditions. Therefore, events associated with changes in proximity conditions can help to define *when* to reconsider current intentions.

Therefore, the key design issues that motivate this study are:

- how to filter relevant belief base updates?

- how to adapt the agent's deliberation cycle in response to changes of these conditions?
- what structures are required to operationalise this behaviour?

I address these questions throughout the study presented in the next sections.

4.3 Design

This section introduces the design of the agent's elements. I integrate the structures that observe changes of the environment to the deliberation process, and demonstrate how they operate when combined together in Section 4.5.

4.3.1 Context Observer

Figure 4.2(A) depicts the architecture of the context observer. In short, the *context observer* works by observing changes of the environment; *calculates* the changes that are relevant for current processing, and *sends events* signaling the occurrences to the planning module. The operations are summarised as follows:

- events related to desire and intention base updates are received by the context observer module[1] and are processed by the (a) *function to extract relevance filters* that populates the *relevance filters base*, whose elements are detailed in the next sub-section;
- in addition, events relating to belief base updates are received by the context observer module[2]; the *(b) function to filter relevant events* selects those events that are relevant to the processing of the existing elements – i.e. desires and intentions – and discards the non-impacting ones. This functionality helps improve the application's performance by reducing the number of operations required to process modifications of the environment. The filtering process is based on relevance filters, stored in a *relevance filters base*;

[1] *Capturing desire and intention base updates*: the functionality to capture events can be operationalised as programming constructs that are implementation dependent; I introduce a practical solution as part of the proof-of-concept implementation in the next chapter.

[2] *Capturing belief base updates*: such as for desire and intention base update events, this functionality is also implemented as a programming construct.

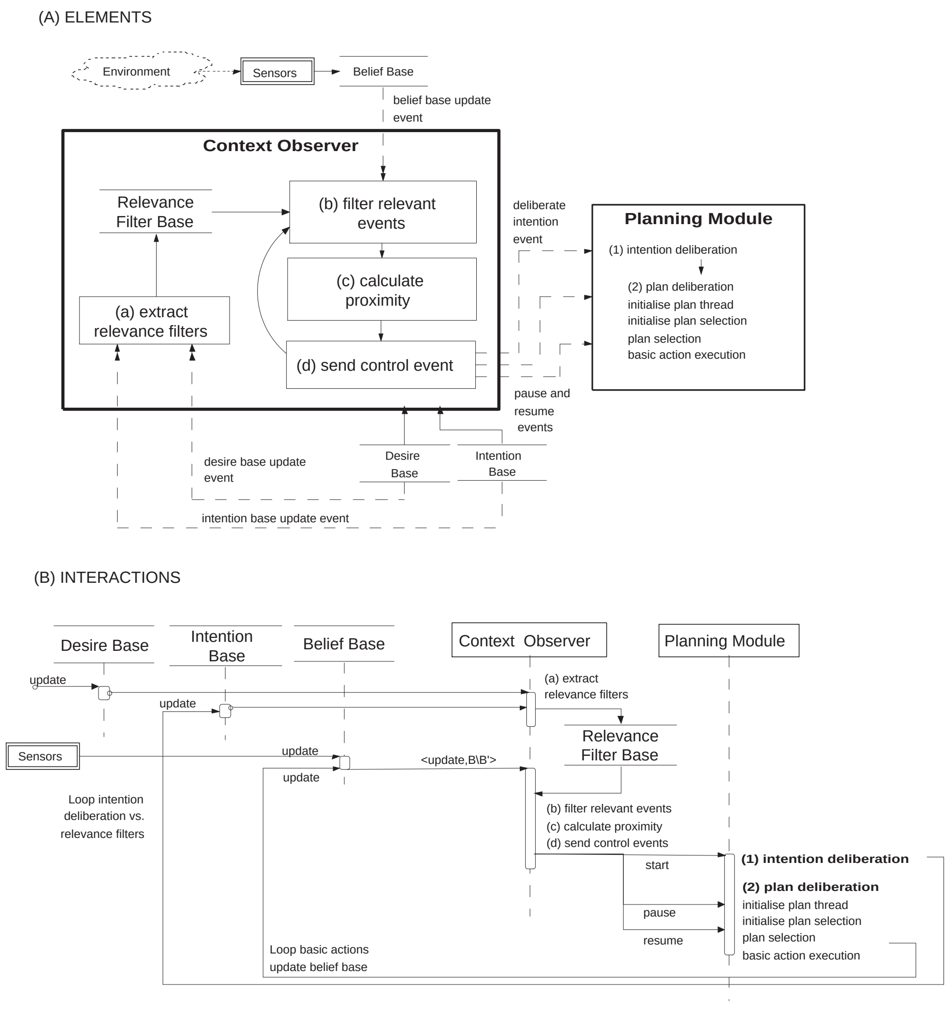


Figure 4.2: Architecture of the Context Observer

- the *(c) function to calculate proximity* implements a second-level processing on the relevant events, calculating their position (i.e. in, near-out, out) in relation to the window of opportunity for a given element;
- the *(d) function to send control events* generates the control events – i.e. start, pause, and resume intention deliberation – and dispatches them to the planning module.

Figure 4.2(B) depicts these interactions. The intention base and desire base components dispatch events related to element updates to the context observer. These events activate the function (a), populating the relevance filter base. Then, updates of the belief base, resulting either from basic action execution or operated by the *sensor* module, are also dispatched to the context observer and activate the function (b), which processes them based on the relevance filters; if they are considered to be relevant, then the module processes the functions (c) and (d), which generates events for the planning module. The process generates new intentions, which ends up updating the intention base and, consequently, a feedback loop to function (a). In addition, the processing of the plan leads to the execution of basic actions that update the belief base, which creates a second feedback loop to function (b). These loops are represented in the diagram.

Relevance Filters

Let us consider that β is an update of the belief base configuration due to the execution of a basic action. That is, if the execution of the action a upon the belief base $\mathcal{B}$ results in the new configuration $\mathcal{B}'$, then $\beta = \mathcal{B}' \backslash \mathcal{B}$. The relevance filter can be defined as below.

Definition 38. (relevance filter)

The relevance filter ρ is a tuple $\langle \beta, E \rangle$, where β is a belief and E is either a desire or an intention.

The *relevance filter base* stores relevance filters as defined below.

Definition 39. (relevance filter base)
$\Upsilon = \{\rho_1, \ldots, \rho_n\}$ *is the relevance filter base where ρ_i is a relevance filter.*

This knowledge base is populated by the (a) *function to extract relevance filters.* It works as follows: whenever a new desire $d = \langle \varphi, C, Redo \rangle$ is asserted in the desire base, the *function to extract relevance filters* adds the tuples $\forall C_i \in C : \langle C_i, d \rangle$ to Υ.

Similarly, every time a new intention $i = \langle \varphi, C \rangle$ is asserted during the course of operations, the process adds $\forall C_i \in C : \langle C_i, i \rangle$ to Υ.

I recall that, depending on the implementation, an agent is a closed system where the desires are defined *a priori*. In these systems, the desire base is static during run-time, thus the filters are populated at startup time only. However, other implementations could allow the manipulation of desires at run-time, for example through either an application programming interface or communication. This definition supports either case.

Exploiting Proximity Conditions

Having the filters populated correctly, then the operation events are processed by the function to filter relevant events as below: for every update of a belief base entry β, if $\langle \beta, E \rangle \in \Upsilon$, then calculate the impact of β over the deliberation of the element E. This functionality is implemented by the *(c) function to calculate proximity.*

Let us assume that the application provides a meta-language to evaluate *proximity conditions.* The implementation of these functions should follow the yes-no questions. I introduce a meta-language to evaluate proximity conditions as defined below to operationalise this evaluation.

Definition 40. (meta-operators to calculate proximity conditions)

- *$in(d)$ and $in(i)$ is true if the condition of the desire d or the intention i evaluates as* in *the window of opportunity;*
- *$near\_out(d)$ and $near\_out(i)$ is true if the condition of the desire d or the intention i evaluates as* near out *the window of opportunity;*
- *$out(d)$ and $out(i)$ is true if the condition of the desire d or the intention i evaluates as* out *of the window of opportunity.*

The implementation must provide programming constructs to operationalise these operations. I introduce a practical solution in the proof-of-concept implementation in the next chapter.

It is possible to explain the relationship between the meta-operator $\mathsf{impact}(.,.,.)$ and the routines to filter relevant events; these operations are defined below.

Definition 41. (operations of meta-operator $\mathsf{impact}(\sigma', \varphi, V_r)$)
Let us consider that β is an update of the belief base configuration due to the execution of a basic action; that is, if the execution of the action a upon the belief base $\mathcal{B}$ results in the new configuration $\mathcal{B}'$, then $\beta = \mathcal{B}' \backslash \mathcal{B}$.

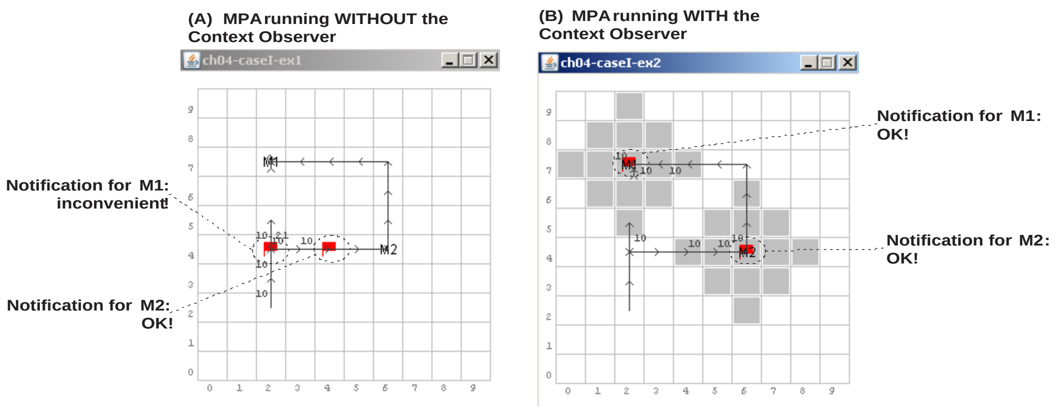

(A) MPA running WITHOUT the Context Observer
ch04-caseI-ex1
Notification for M1: inconvenient!
Notification for M2: OK!
M1
M2
(B) MPA running WITH the Context Observer
ch04-caseI-ex2
Notification for M1: OK!
Notification for M2: OK!

Figure 4.3: Case Study

- impact$(\beta, \varphi, start) \iff \langle\beta, \langle\varphi, C, R\rangle\rangle \in \Upsilon \wedge in(\langle\varphi, C, R\rangle)$; *that is, the configuration β leads to the execution of the desire $\langle\varphi, C, R\rangle$ iff the condition of the desire evaluates as* in *the window of opportunity. Note: as a follow up of this operation, the deliberation cycle will execute the operation for intention deliberation, which will be successful if not already executing a planning thread for the deliberation of φ.*

- impact$(\beta, \varphi, pause) \iff \langle\beta, \langle\varphi, C\rangle\rangle \in \Upsilon \wedge \langle run, \varphi, \pi, \Theta\rangle \in \mathcal{P} \wedge out(\langle\varphi, C\rangle)$; *that is, the configuration β leads to pausing the execution of the running planning thread iff the condition of the intention evaluates as* out *of the window of opportunity.*

- impact$(\beta, \varphi, resume) \iff \langle\beta, \langle\varphi, C\rangle\rangle \in \Upsilon \wedge \langle pause, \varphi, \pi, \Theta\rangle \in \mathcal{P} \wedge in(\langle\varphi, C\rangle)$; *that is, the configuration β leads to resuming the execution of the paused planning thread iff the condition of the intention evaluates as* in *of the window of opportunity.*

Finally, the events to start, pause, and resume processing thread are asserted to the event base as part of the *(d) function to send control events.*

Advantages and Limitations

To demonstrate the advantages of the proposed model, I introduce a comparative analysis of the qualitative performance gain in executing the same application, running on the same environment. I recall the example introduced in Chapter 1. Let us consider that the mobile user is carrying a mobile personal assistant (MPA) with the functionality described below.

> A user has a mobile application configured to download information for upcoming meetings. Let us say that there is a meeting

> about Project A booked for 10 am at office A110 (goal M1). At 9:53 am, when the user is within 2 minutes walking distance from office A110, the application starts to download the information related to the project (position (i)). However, while walking towards the meeting room, the user detours to a sideline meeting with his boss (position (ii)). There, the application exchanges notes with the boss' mobile assistant (goal M2). Depending on how this conversation evolves, the user might decide to cancel his participation in the meeting. In this case, he moves in another direction (position (iv)). The application should deliver the information only if the user is in the right context.

This application is able to sense its current position and time, handle the user's agenda inferring for upcoming meetings, notify the user about upcoming meetings, negotiate information with other agents, retrieve information from a server, and notify the user about the received files. The application can download information at the speed of one piece per square-move. Finally, let us assume that the MPA knows it needs to download four pieces of information for meeting M1 (coded as a belief in the belief base).

The scenario in Figure 4.3(A) presents the results of the application with the context observer module disabled, where the application behaves following the traditional "check $\rightarrow$ deliberate $\rightarrow$ act" loop. The goal $M1$ is asserted to the board when the user is at position (2,2). As the user starts to walk toward $M1$, the MPA deliberates the intention to have the information for that meeting and starts the execution immediately. When the user is at position (2,5) the goal $M2$ is asserted to the environment and the user detours to meet his boss. The MPA infers the opportunity to exchange information with the boss' MPA and starts the downloading process in parallel to that for M1. The MPA completes the download of information for $M1$ when the user is at (2,4), delivering the notification at that point (first flag on the board). However, at this position the user is out of the window of opportunity for M1. Hence, information about $M1$ is inconvenient and out of context at this point. Moreover, if after reaching $M2$ the user decides to move elsewhere (i.e. the user decides to no longer participate in meeting $M1$), the information was already inconveniently delivered and the resources spent retrieving the information were wasted. This situation happens because the deliberation system does not observe the context while processing the plan to retrieve and display the information;. that is, it continues to process the adopted plans no matter what.

The scenario in Figure 4.3(B) is the same environment and application as the scenario above, but now the agent has the context observer enabled. In this case, it processes the environmental events related to the windows of opportunity, which is calculated as the Manhattan distance ($\Delta_x + \Delta_y$) from the goal's position. The processing of the plan to download the information for meeting M1 starts only when the user steps into the window of opportunity for that goal at square (2,5). This is a result of the meta-operator $\mathsf{impact}(\beta, \varphi_{m1}, V)$, where β reflects the update of current position and φ_{m1} is the goal to retrieve the information for M1. The agent triggers $V = start$ asserting the intention $\langle \varphi_{m1}\theta, C\rangle$ to $\mathcal{I}$. However, as the goal M2 is asserted to the environment and the user detours back to (2,4) towards M2, the meta-operator $\mathsf{impact}(\beta, \varphi'_{m1}, V)$, where $\varphi'_{m1} = \varphi_{m1}\theta$, evaluates $V = pause$ and as there is a planning thread for φ'_{m1} in running state in $\mathcal{P}$ (resulting from the intention execution), it implies the transition: $\mathcal{P} : \langle run, \varphi'_{m1}, \pi, \Theta\rangle \rightarrow \langle pause, \varphi'_{m1}, \pi, \Theta\rangle$.

This planning thread will remain paused until the user steps back into the window of opportunity for M1, at position (4,7), when $\mathsf{impact}(\beta, \varphi'_{m1}, V)$ evaluates $V = resume$ and as there is a paused planning thread for φ'_{m1} (from the operation above) it now implies the transition: $\mathcal{P} : \langle pause, \varphi'_{m1}, \pi, \Theta\rangle \rightarrow \langle run, \varphi'_{m1}, \pi, \Theta\rangle$.

It is assumed that the implementation provides the operations to check the consistency of the intention with regards to resources and other possible conflicting intentions adopted while that intention was paused.

Therefore, the solution with the context observer provides a better qualitative performance in this situation, as it avoids delivering information outside the window of opportunity for effectively using it. Moreover, should the user decide not to participate in M1 after the meeting in M2 (i.e. to move to position (iv) in the illustrative example), the application will not continue to retrieve the information, saving resources and avoiding the delivery of useless (and therefore incoherent) information.

This simple test scenario shows the qualitative performance advantage of the extended deliberation model. I note that similar benefit could be achieved by coding the proper conditions in the guards of the plans, however, that approach would be at the expense of programming efforts. The proposed model achieves the benefits as a result of an inherent feature of the deliberative behaviour, making the application easier and more intuitive to build.

The performance advantage is highlighted for applications that have large number of desires and concurrent intentions, higher degree of dynamism, and larger belief bases. In Chapter 6, I introduce case studies where I compare the performance of different configurations.

This scenario also explains how the proposed method helps to resolve the key questions when handling dynamic environments, i.e.:

- How to handle changes in the environment?
- How to react timely these to changes?
- How to adapt and keep the application coherence?

In short, the context observer filters the relevant changes, that is the ones that can impact current processing, and evaluates proximity conditions to coordinate the processing. The application has its performance enhanced by exploring events of the environment and adapting the deliberation cycle accordingly. In addition, this feature contributes to application's consistency and coherence, for example, by avoiding waste of resources and the delivery of out-of-context information.

An obvious limitation of the approach is that it requires the programming structures to evaluate the events relating to the windows of opportunity. As mentioned previously, this is a domain-specific problem. Thus, reductions are possible if only certain parameters of the context are considered, such as time and space. The method is only as good as the capability to precisely evaluate these events in real-time, however; a potentially complex and resource-consuming task in realistic mobile services. I introduce a practical solution as part of the prototype in the next chapter.

4.3.2 Planning Module

In this section, I explain the elements and interactions required to operationalise the processing; complementing the pending descriptions in the previous chapter.

Figure 4.4 depicts the architecture of the *planning module*. This module bridges reactive- and goal-oriented deliberative behaviour. In the foreground it works triggered by events from the context observer module, what is essentially a reactive behaviour; however in parallel, it executes the plan to achieve the goals, which is essentially a goal-oriented behaviour. The architecture is composed of three processing units:

1. the *sub-routine for intention deliberation*, which processes events to generate intentions incoming from the context observer; the operations are detailed in Section 3.4.1;

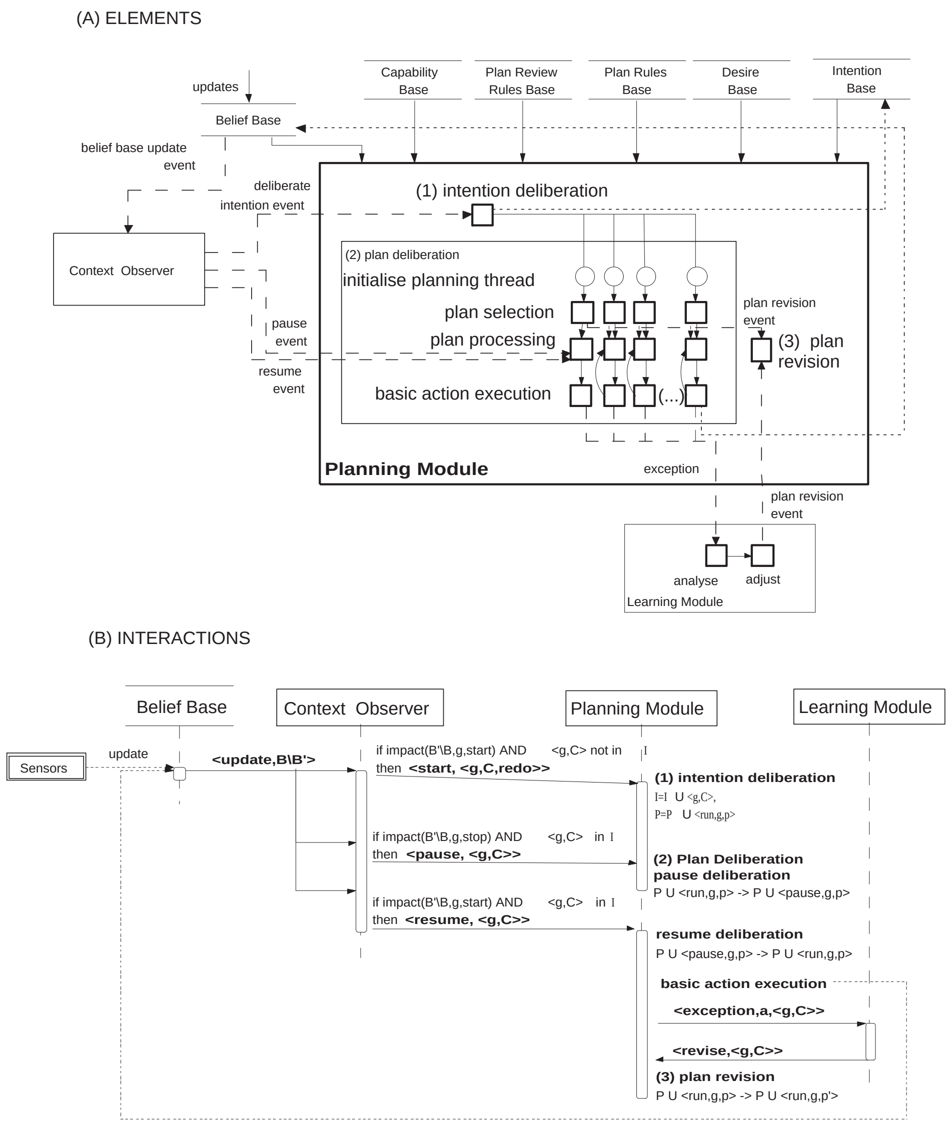


Figure 4.4: Planning Module

2. the *sub-routine for plan execution*, which implements the steps for planning thread initialisation, plan selection, and basic action execution. I recall that several planning threads can execute in parallel, which is represented in the diagram; the operations are detailed in Section 3.4.2, and;

3. the *sub-routine for plan revision*, which implements the operations detailed in Section 3.4.2. I recall that besides *reactive plan revision*, it is also possible to have *proactive plan revision* based on current execution context.

Figure 4.4(B) depicts the interactions between the elements. I note that the interactions are started by either updates in the belief base or exceptions in the basic action execution. The former fires the event $\langle update, \mathcal{B}' \rangle$ which reports the updated model of the world. These events are processed by the context observer module and dispatched to the appropriate planning module. These interactions are described below:

- if the context observer processes a belief update such that $\mathsf{impact}(\beta, \varphi, start)$ and there is no intention for φ yet, then the event $\langle start, \langle \varphi, C, redo \rangle \rangle$ is activated, which triggers the (1) *sub-routine for intention deliberation.* If a belief base update impacts the processing of a desire, and there is no intention to achieve the desire's goal being processed yet, then the application commits to that goal.

- if the context observer processes a belief update such that $\mathsf{impact}(\beta, \varphi, pause)$ and there is an intention for φ, then the event $\langle pause, \langle \varphi, C \rangle \rangle$ is activated, which triggers the (2) *sub-routine for plan execution*, described above. Internally, this event will force the pausing of the processing thread, as per Definition 36. If a belief base update impacts the processing of a current intention, such that the intention should be paused, the application sends an event to the plan execution routine to pause.

- if the context observer processes a belief update such that $\mathsf{impact}(\beta, \varphi, resume)$ and there is an intention for φ, then the event $\langle resume, \langle \varphi, C \rangle \rangle$ is activated, which triggers the (2) *sub-routine for plan execution*, requesting the resuming of a processing thread.

Next, there are the interactions between the *sub-routine for plan execution*, the *learning module*, and the *sub-routine for plan revision. Unexpected events* can lead to exceptions during basic action execution; that is an execution fails to result produce an expected situation. Hence, exceptions in basic action execution can also initiate the events $\langle exception, \langle \varphi, C \rangle \rangle$, which are captured by the *learning module.* After processing, this module fires back the event $\langle revise, \langle \varphi, C \rangle \rangle$, which triggers the (3) *sub-routine for plan revision.* This sub-routine revises the current plan π into a new form π' and continue the processing.

4.3.3 Learning Module

In Section 3.5.3, I suggested the learning module provides the methods to

1. receive events from the planning module about exceptions during basic action execution;
2. analyse the events;
3. implement the logic to adjust the operation values that could influence the processing, and;
4. request plan revision.

For the first item, the learning module receives exceptions in the format $\langle exception, \langle \varphi, C \rangle \rangle$. The *function to analyse operation parameters* implements the logic to correlate the attributes of the current operational environment – i.e. beliefs, desires, other intentions, planning rules – with possible causes for the exception, before calling the *function to adjust operation values*, which promotes the revision of the existing entries.

In Section 3.5.3, I argued that issues of *what* and *how* to learn are out the scope of this work. Although I am not detailing the operational aspects of these functions, they should be provided as *abstract interfaces* that can be overloaded[3] during the implementation, and I introduce a practical solution for such programming constructs in the next chapter.

I also introduced a number of possible run-time adjustments that could be implemented by the learning process, such as "adjust conflict rules", "adjust prioritisation rules", and "adjust cost rules". More elaborate learning techniques could review existing plan rules, desires, and other belief base entries.

In the next section, I put the pieces together and analyse the extended deliberation cycle.

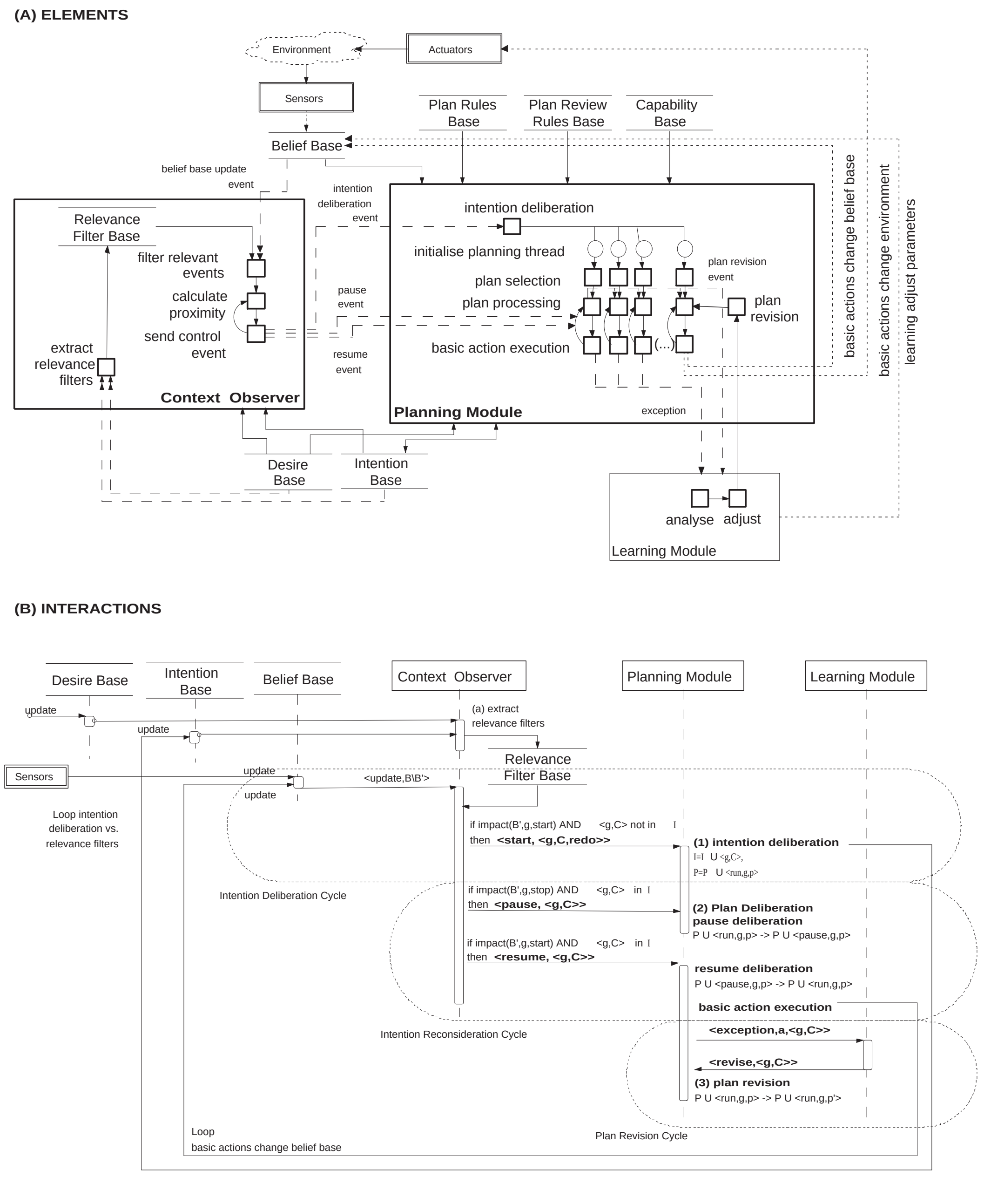


Figure 4.5: Complete Design Model

4.3.4 Putting the Pieces Together

Figure 4.5(A) depicts the architecture. I highlight that the interaction between modules works through events in a cascading architecture, starting from updates to the knowledge bases.

For instance, when the conditions are adequate a desire is deliberated forming an intention. This process implies the update of the relevance filters base by the context observer. If the goal is reached, then the intention is retracted from the intention base. Consequently, those monitoring filters are no longer relevant and must also be retracted.

I introduce the deliberation cycle in this model, aiming to explain how the structure works when the pieces are combined. I also aim to demonstrate how the inference system is now equipped to support the issues outlined in the previous chapter.

For the sake of this presentation, let us consider an agent $\langle \mathcal{B}_0, \mathcal{D}, \emptyset, \emptyset, \mathcal{A}, \varphi, \emptyset \rangle$, where the elements are defined below.

- $\mathcal{D}$ contains the set of desires $\{\langle \varphi_1, C_1, redo_1 \rangle, \ldots, \langle \varphi_n, C_n, redo_n \rangle\}$;
- φ contains the set of plan rules $\{\varphi_1^1 \leftarrow C_1^1 \mid \pi_1^1, \ldots, \varphi_1^j \leftarrow C_1^j \mid \pi_1^j, \ldots, \varphi_n^1 \leftarrow C_n^1 \mid \pi_n^1, \ldots, \varphi_n^k \leftarrow C_n^k \mid \pi_n^k\}$;
- $\mathcal{A}$ contains set of capabilities $\{\langle C_1, \mathsf{Act}_1, C_1' \rangle, \ldots, \langle C_m, \mathsf{Act}_m, C_m' \rangle\}$, and;
- $\mathcal{B}_\prime$ contains the initial knowledge of the world and includes information about conflict and prioritisation rules.

In addition, the relevance filter base gets populated during the agent initialisation process. Once the filters are populated, the configuration of the relevance filter base is as follows.

$\{\langle \beta_1, \langle \varphi_1, C_1, redo_1 \rangle \rangle, \ldots, \langle \beta_i, \langle \varphi_i, C_i, redo_i \rangle \rangle, \ldots, \langle \beta_n, \langle \varphi_n, C_n, redo_n \rangle \rangle\}$, where $\mathcal{D} \models \langle \varphi_i, C_i, redo_i \rangle$, and $c_i \in C_i$.

However, let us assume that the set of entries in $\mathcal{B}_\prime$ do not imply the processing of any of the desires in $\mathcal{D}$. That is, none of the conditions of the desires $d_i \in \mathcal{D}$ are satisfied by the entries in $\mathcal{B}_\prime$. In this case, obviously the conditions will be satisfied once the entries in the belief base reach a state that activate any of the desires $d_i \in \mathcal{D}$. This vision implies that changes of the environment, perceived through *sensors*, activate the processing. This cycle is introduced below.

[3] *Overload*: in the sense on object-oriented programming where an method can be overloaded by a child class.

Intention Deliberation Cycle

The cycle for intention deliberation starts with a modification of a monitored entry β. I recall that the belief base is updated by either the sensors or the execution of basic actions. The update of the belief base initiates the event $\langle \mathit{update}, \beta \rangle$, which is captured by the *context observer* and processed by the *function to filter relevant events.*

If a relevance filter in Υ implies $\langle \beta, \langle \varphi, C, \mathit{redo} \rangle \rangle$ and the evaluation of $\mathsf{impact}(\beta, \varphi, V)$ results in $V = \mathit{start}$, and the intention base $\mathcal{I}$ does not imply a planning thread for φ, then the context observer fires the event $\langle \mathit{start}, \langle \varphi, C, \mathit{redo} \rangle \rangle$.

In response to this event, the planning module executes the *sub-routine for intention deliberation.* From this point, there are three possible course of actions:

1. there are no current conflicting intentions, in which case a new intention is asserted to the intention base (see Definition 21);

2. there is a current conflicting intention and the new one has lower priority, in which case the new intention is created but the status is assigned as "paused" (see Definition 22), or;

3. there is a current conflicting intention and the new one has higher priority, in which case the inference system promotes *goal switching*; pausing the current intention and asserting φ to the intention base (see Definition 23).

This cycle encompasses the following elements and interactions represented in the diagram in Figure 4.5(B):

> belief base update event $\rightarrow$ sub-routine to filter relevant events $\rightarrow$ sub-routine to calculate proximity $\rightarrow$ sub-routine to send control event (these last three sub-routines are part of the context observer) $\rightarrow$ intention deliberation event $\rightarrow$ sub-routine to deliberate intentions.

Advantages. The main advantage of this design is its computation simplicity. That is, the intention deliberation operates in reaction to changes of the environment. The filters in the context observer optimise the number of operations to verify the desires' conditions, reducing processing load.

I compare these operations against the *strictly sequential deliberation* of the simple BDI deliberation model, which is implemented by several agent languages. The process (i) verifies the environment, (ii) selects the possible options to deliberate, (iii) forms intentions based on the selected options, (iv) selects an intention with the higher priority, and (v) finally, it forms plans to execute the selected intention. As mentioned, a shortcoming of the basic deliberation model is that it does not consider the impact of belief base updates on in-processing elements. The step that would reconsider current processing (i.e. *(b) reconsider current intentions*) happens only after executing current plans (i.e. *(e) execute plans*).

To address this limitation, the proposed model promotes the separation between the activation method and deliberation process, allowing the processing of concurrent processing, i.e. the context observer is continually monitoring the environment for desires' condition while other intentions are being processed by the planning module. This feature helps to *balance reactiveness and goal-oriented behaviour* in the overall inference system; another shortcoming of the simple BDI model.

Plan Execution Cycle

The cycle for plan execution starts with the planning module executing the *sub-routine for intention deliberation.* It is formed by the following elements and interactions represented in the diagram in Figure 4.5(A):

> sub-routine to deliberate intentions → initialise planning thread → plan selection → plan processing → basic action execution, and loop back to plan selection until either the goal has been achieved or the intention has been reconsidered

Multiple execution threads can be executed in parallel, as long as they do not conflict on resources. I note that this cycle is essentially goal-oriented, if one does not consider the possibility of interleaving processing review. In the simple BDI model, Rao and Georgeff [1995] proposes to interleave intention reconsideration and plan execution. That is, the process of *(b) reconsidering current intentions*) happens only after executing current plans (i.e. *(e) execute plans*). However, there are situations where the reconsiderations must take place *between* the plan execution process, namely due to *unexpected events* and *adaptation.* The next two cycles complement this cycle by interrupting it when events of the environment suggest the opportunity to do so.

Advantages. The advantage of the proposed design is that multiple plan execution cycles can run in parallel separated in *planning threads.* This structure was introduced in Section 3.4.2, representing *self contained* planning processes that can be controlled individually. Control is implemented by changing the operation status. The *paused status* means interrupting the execution operations but saving the current status, that is the current goal, unification stack, selected plan rules, and other related parameters. That way the operation can be *resumed* when (and if) the conditions are favourable; a process that happens in response to conflict resolution and goal switching, as presented in Section 3.4.1. This feature allows the efficient implementation of *deliberation checkpointing*; that is the ability to start, pause, and resume processing threads. A desirable feature of the design, it also promotes *processing optimisation* and helps to support the concept of adaptation in response to events with the window of opportunity.

Intention Reconsideration Cycle

Firstly, I keep the distinction between *intention revision*, which is the act to reconsider the intention to achieve a given goal – e.g. as part of conflict resolution, prioritisation process, and goal switching – and *plan revision*, which is the act to revise the steps being adopted to achieve a given goal, as introduced in the next topic. In this work, I introduce a method for intention revision based on *internal and external events*. The deliberation cycle provides the structures to verify conflicts and prioritisation for goal resolution for the *internal events*, as described in Section 3.4.1. In addition, the context observer controls the *reconsideration process* initiated by relevant changes of the environment, that is in response to *external events*.

I highlight the close relationship between the resulting values and events relating to the window of opportunity.

It works as follows. The belief base update operation fires the event $\langle \mathit{update}, \beta \rangle$, which is captured by the *context observer* and processed by the *function to filter relevant events*. If an relevance filter in Υ implies $\langle \beta, \langle \varphi, C \rangle \rangle$ and the evaluation of $\mathsf{impact}(\beta, \varphi, V)$ results in $V = \mathit{pause}$, then the context observer fires the event $\langle \mathit{pause}, \varphi, \pi \rangle$, where $\mathcal{I}$ implies $\langle \varphi, C \rangle$. On the other hand, if $\mathcal{P}$ implies $\langle \mathit{pause}, \varphi, \pi \rangle$ and $\mathsf{impact}(\beta, \varphi, V)$ results in $V = \mathit{resume}$, then the context observer fires the event $\langle \mathit{resume}, \langle \varphi, C \rangle \rangle$.

Therefore, the intention revision cycle is composed of the methods to process belief base updates, filter the relevant ones relating to existing executions, calculate the proximity conditions, and notify the plan formation process to pause and resume the processing thread in order to optimise the application's behaviour. This process is formed by the following elements and interactions represented in the diagram in Figure 4.5(B):

belief base update event $\rightarrow$ sub-routine to filter relevant events

> → sub-routine to calculate proximity → sub-routine to send control event (these last three sub-routines are part of the context observer). At this point, it can either send *pause event* to the plan formation process → change status of the planning thread to *paused*, or send *resume event* to the plan formation process → change status of a paused planning thread to *running*

Advantages. The intention reconsideration cycle is essentially a *reactive* operation, whose actions are triggered in reaction to changes of the environment. Again, the filters in the context observer optimise the number of operations to verify the desires' conditions, reducing processing load. The design is simple and efficient. The context observer filters the relevant changes, which are the ones that can impact current processing threads, and evaluates proximity conditions to coordinate the processing. The application has its performance enhanced by exploring events of the environment and adapting the deliberation cycle accordingly. This feature contributes to the application's *consistency* and *coherence*.

The optimisation is a consequence of taking advantage of environmental information to decide *when* to reconsider. The operations are facilitated by the ability of the planning process to handle concurrent processing and support processing checkpoint.

Plan Revision Cycle

Plan revision is the act to revise the steps being adopted to achieve a given goal. It is formed by the interactions between the *sub-routine for plan execution*, the *learning module*, and the *sub-routine for plan revision*. Section 3.4.2 explains the conditions, operations, and consequences of the plan revision process.

I recall that this operation is activated by exceptions during basic action execution. These exceptions are captured by the learning module, which promotes the analysis of the conditions and required adjustments, before requesting the plan revision. I proposed a centralised *sub-routine for plan revision* in order to facilitate coordination.

This process is formed by the following elements and interactions represented in the diagram in Figure 4.5(B):

> processing conditions or *exceptions* executing basic actions → learning module – analyse the situation and promotes adjusting of processing conditions accordingly → sub-routine for plan review –

changes the plan execution process → loop back to the plan formation process

Advantages. The plan revision process is an essential component in the BDI-model deliberation cycle. It allows the application to recover from unexpected situations and, thus, enhances the application's stability.

In this model, the *learning module* promotes utility learning. That is, the process of analysing and adjusting the execution parameters is triggered by an exception event, resulting from a failure to execute a basic action. One can claim that this approach is *self-adjustment* and not real learning, in the sense of *self-programming.* Although, as proposed in Section 3.5.3, this topic is out of the scope of this work, I consider it sufficient to conceptualise the operations of the learning module and to provide the structures required for its operations.

Therefore, the combined operations of the interleaving processing threads help to support the requirements for mobile applications' development, as suggested in Section 2.5, that is: impromptu reconsideration of in-processing elements; take advantage of events from window of opportunity; maintain application coherence amidst environmental instabilities, and; optimise resource utilisation.

4.4 Conclusion

This chapter introduced the *design model* that describes solutions for the pending issues of the conceptual model, such as the elements taking advantage of environmental information and promoting performance enhancement. This model's main contribution is to exploit features of the environment – *viz* events relating to the window of opportunity – to infer *when* to revise individual processing threads. In addition, it describes *how* to implement applications equipped to adapt its processing to cope with environmental instabilities. As presented in Chapter 2, this is one of the shortcomings in current developments.

In the proposed solution, the applications are equipped to sense and filter relevant changes with the situation and checkpoint the current processing, thus avoiding delivering out-of-context information and improving the application's performance. In the end of the next chapter, I demonstrate how this technique provides a solution for different application domains.

Different agent platforms could support the proposed design if they are adapted to do so. The design could be integrated in field technologies to support practical solutions. However, the design introduces new requirements that are not commonly available in existing implementations: namely, concurrent intention execution, deliberation checkpointing, and the possibility of exploring environmental events for intention reconsideration.

The proposed model can also be applied to implement applications in different knowledge domains. The benefits would apply in any domain in which software applications must operate in a dynamic environment, use context information for decision making, and implement both reactive and proactive behaviour. The main difference is that the context observer needs to be equipped to realise and make sense of different dimensions of context information.

In the next chapter, I progress detailing the operations and introduce a proof-of-concept implementation.

Chapter 5

Prototype

"Like a beautiful flower that is colourful but has no fragrance, even well spoken words bear no fruit in one who does not put them into practice."
Buddha.

This chapter presents a proof-of-concept implementation of the proposed model.

5.1 Introduction

In the previous chapters, I introduced an extended BDI-model deliberation cycle that takes advantage of environmental events to enhance the application's performance.

I suggested that different agent platforms could support the proposed design if they are adapted appropriately. The design could be integrated with field technologies to support practical solutions. Nonetheless, it introduces new requirements concurrent intention deliberation, processing checkpoint, and the using events within the environment for intention reconsideration that are not commonly available in existing implementations.

In this chapter, I present a proof-of-concept implementation. I am extending an existing agent platform; remodelling its inference system, and re-using the support provided by the basic language. The goal of this exercise is two-fold: firstly, to demonstrate that the specifications are reproducible as software applications, and; secondly, to provide a practical solution for the technical issues mentioned in the previous chapters.

The chapter is organised as follows. Section 5.2 analyses the support existing agent languages offer and explains the choice for the *2APL Practical Agent Language* in this work. Section 5.3 presents the integration and implementation efforts. Section 5.4 introduces the practical solutions adopted in this project. The chapter concludes with a discussion of achieved results and extensibility in Section 5.5.

5.2 Analysis

In order to facilitate the development effort, the supporting agent programming language should be a BDI-model language that provides the "common elements" in the BDI-model, such as the representation for beliefs, desires (or goals), intentions, plans, basic actions, and the operational semantics for these elements[1]. I suggest that this integration is desirable to promote the re-usability of existing specifications and progress towards field acceptance. In Section 2.4.4, I introduce a number of existing BDI-model agent programming languages. I list the potential candidates and analyse their features below:

- *AgentSpeak(L)*, presented by Bordini et al. [2002], and its implementation *Jason*, introduced in Bordini et al. [2007], provides explicit representation of the common elements. In addition, it provides meta-language operations to access the deliberation cycle functionality and implements the concept of "processing coordinated by events", which is related to the proposed definitions.

- *Lightweight Extensible Agent Platform* (LEAP), presented in Bergenti et al. [2001], also provides the explicit representation of common elements, although support for processing coordination is not directly available. Being an open source platform, it is possible to access, manipulate, and extend its programming interface.

- *JACK*, presented in Busetta et al. [1999] and [Padgham and Winikoff, 2002], supports the common elements, although beliefs and goals lack a formal semantics. It does however incorporate the concept of event-based coordination, which can be extended to support the proposed functionality.

- *3APL*, introduced in Hindriks [2001], provides the programming constructs to implement beliefs, goals, and capabilities. It also uses practical reasoning rules to generate, update, and revise plans. Executed by an *interpreter* that deliberates on the cognitive attitudes, the program's

[1] *Code re-usability*: of course, if the developer can access and re-use (completely or partially) the existing processing structures, then development effort is further simplified.

operations are simple and well defined, allowing their extension by programming constructs. The current implementation, however, does not provide the programming interface to access the internal functionality.

- *3APL-M*, introduced in Koch et al. [2005], is an implementation of 3APL targeting mobile devices. It inherits the operational semantics from 3APL and adds the meta-operators to control the deliberation cycle functionality, therefore I argue that its code base could be easily adapted to support the extended functionary.

- *2APL*, introduced in Dastani et al. [2007], is a BDI-based agent-oriented programming language that supports the common element, and provides the operational semantics to support declarative concepts such as belief and goals and imperative style programming (such as events and plans). The current implementation is built on top of the JADE platform, thus it inherits the support to multi-agent communication and coordination.

In this work, I selected the *2APL agent programming language* as the basis for the prototype. There are a few compelling reasons for this choice, as described below.

Firstly, the 2APL language provides the "common elements" of the BDI architecture. That is, it defines a clear and simple syntactic representation of the basic elements – i.e. beliefs, goals, and plans. This is advantageous, as these structures can easily relate to the syntax described in Section 3.3.

Dastani et al. [2007] introduces a formal semantics of 2APL in terms of a transition system. That work describes the operations of the 2APL deliberation cycle (or the *interpreter*), which facilitates the process of mapping of the operations proposed in Section 3.4.

The 2APL language offers an *exception handling* mechanism. Related to the syntax for programming constructs for planning and plan repairing in the proposed model, this mechanism allows the programmer to specify how an agent should repair its plans. Although similar to the plan revision rules introduced in 3APL the 2APL rules can only be applied to repair failed plans, as described in Dastani et al. [2006].

Finally, the 2APL language supports events and plans, and promotes the effective integration of programming constructs with imperative style programming. It also provides the concept of *events* for notifying environmental changes. These elements facilitate integrating the *context observer module* to the deliberation cycle.

5.2.1 On 2APL

Having explained the rationale behind adopting the 2APL agent programming language in this work, I briefly introduce the language in this sub-section. I quote the following statement from the *2APL User Guide*[2]:

> *2APL (pronounced as double-a-p-l) is an agent-oriented programming language that facilitates the implementation of multi-agent systems. At the multi-agent level, it provides the programming constructs to specify a multi-agent system in terms of a set of individual agents, a set of environments in which they can perform actions, and the access relation between the individual agents and the environments. At the individual agent level, it provides the programming constructs to implement cognitive agents based on the BDI architecture. In particular, it provides the programming constructs to implement an agent's beliefs, goals, plans, actions (such as belief updates, external actions, or communication actions), events, and a set of rules through which the agent can decide which actions to perform. 2APL supports the implementation of both reactive and pro-active agents with adjustable autonomy and decision process.*

The 2APL agent configuration is defined as:

$$A_i = \langle i, \sigma, \gamma, \Pi, \theta, \xi \rangle$$

Where i is a string representing the identifier, σ is a set of belief expressions representing the *belief base*; γ is a list of goal expressions representing the *goal base*; Π is a set of plan entries enriched with additional information, representing the plan base; θ is a ground substitution that binds domain variables to ground terms, and; ξ is the event base.

Each plan entry is a tuple (π, r, p) where π is the executing plan, r is the instantiation of the *PG-rule* through which π is generated, and p is the plan identifier. The event base ξ is a tuple $\langle E, I, M \rangle$ where:

- E is a set of events received from external environments; an event has the form $event(A, S)$, where A is a ground atom passed by the environment S;

[2] *2APL User Guide*: available from the project's web-site at http://www.cs.uu.nl/2apl, accessed in Jul-2008.

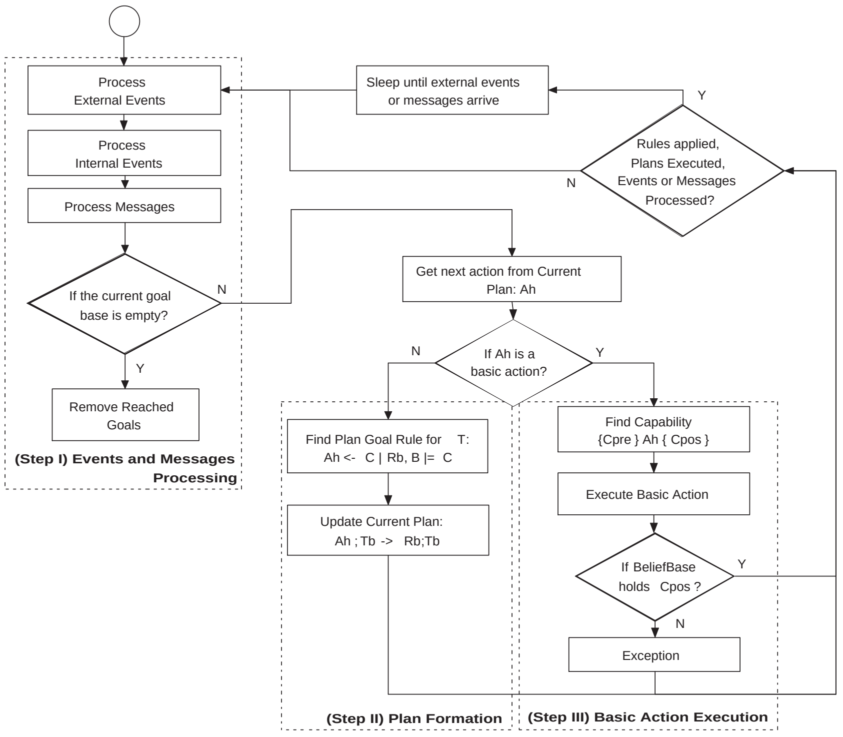

Process External Events
Process Internal Events
Process Messages
If the current goal base is empty?
N
Y
Remove Reached Goals
(Step I) Events and Messages Processing
Sleep until external events or messages arrive
Y
Rules applied, Plans Executed, Events or Messages Processed?
N
Get next action from Current Plan: Ah
If Ah is a basic action?
N
Y
Find Plan Goal Rule for T: Ah <- C | Rb, B |= C
Update Current Plan: Ah ; Tb -> Rb;Tb
Find Capability {Cpre } Ah { Cpos }
Execute Basic Action
If BeliefBase holds Cpos ?
Y
N
Exception
(Step II) Plan Formation
(Step III) Basic Action Execution

Figure 5.1: 2APL Deliberation Cycle

- I is a set of identifiers denoting failed plans; an identifier represents exceptions thrown because of a plan execution failure;
- M is the set of messages sent to the agent; each message is of the form $message(s, p, l, o, \phi)$, where s is the sender identifier, p is a performative, l is the communication language, o is the communication ontology, and ϕ is a ground atom representing the message content.

The platform provides the deliberation cycle (or *interpreter*) to animate the agent. The operational semantics for this process is presented in Dastani et al. [2007]. The operations are segmented in three steps as depicted in Figure 5.1 and explained below:

- *(Step I) Events and Messages Processing*, processes the existing events and incoming messages from the event base; the rules are defined by the *procedure call rules* (PC-Rules);
- *(Step II) Plan Formation*, processes the plan reasoning rules (PR-Rules), and;
- *(Step III) Basic Action Execution*, executes the physical and mentalist actions during plan execution.

I describe these elements in detail in the next section when I introduce the prototype's deliberation cycle.

5.2.2 Shortcomings of 2APL

I argue that the current 2APL implementation lacks some features required for the purpose of this work. For example, it does not provide the structures to support concurrent intention processing or impromptu review of the intention execution thread. Consequently, it does not provide the definitions for prioritisation and conflict rules. In addition, there is no clear support for qualitative plan selection in 2APL's deliberation process; that is, the semantics of 2APL do not specify the selection criterion when multiple plan rules are possible. The proposed model introduces the definition of *cost rules* and the operations to calculate costs as part of the plan selection function.

Moreover, although the 2APL definitions provide the operational semantics for plan revision. The specifications describe that plan failures triggers the plan revision process. In the proposed model, the deliberation process takes environmental events in consideration to implement pro-active plan revision – i.e. to promote plan revision before it fails.

I detail the implementation and describe how it covers these limitations in the next section.

5.3 Implementation

Figure 5.2 provides the prototype's high-level architecture. It presents the main components: knowledge databases, processing modules, and the interfaces to field technologies. Table 5.1 complements that presentation and summarises the object representations in the implementation. These definitions imply the presence of a PROLOG engine that provides the logical basis for the elements.

The implementation also provides the programming interface for incoming events (the box labelled as "EvtIn" in the diagram) and the outgoing events (the box labelled as "EvtOut"). These structures coordinate event-passing between the modules.

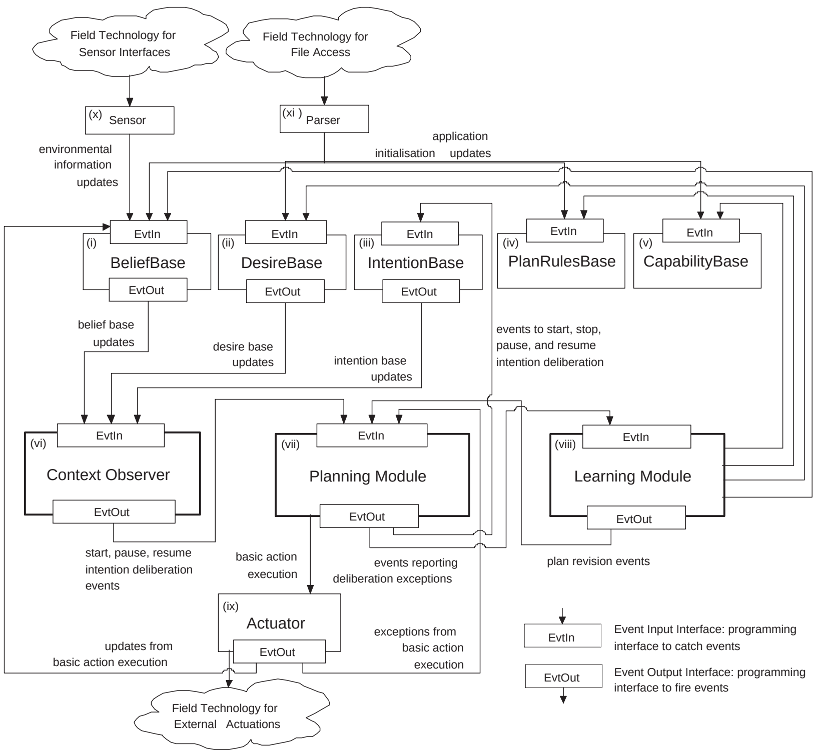

Field Technology for Sensor Interfaces
Field Technology for File Access
(x) Sensor
(xi) Parser
environmental information updates
application initialisation updates
EvtIn
(i) BeliefBase
EvtOut
(ii) DesireBase
(iii) IntentionBase
(iv) PlanRulesBase
(v) CapabilityBase
belief base updates
desire base updates
intention base updates
events to start, stop, pause, and resume intention deliberation
(vi) Context Observer
(vii) Planning Module
(viii) Learning Module
start, pause, resume intention deliberation events
basic action execution
events reporting deliberation exceptions
plan revision events
(ix) Actuator
updates from basic action execution
exceptions from basic action execution
Event Input Interface: programming interface to catch events
Event Output Interface: programming interface to fire events
Field Technology for External Actuations

Figure 5.2: Architecture of the Prototype Application

Element	Object Representation
Belief	PROLOG Term
Goal	PROLOG Term
Condition	Array of PROLOG Terms
Desire	< Condition, Goal, redo >
Intention	< status, Condition, Goal, PlanningThread >
PlanningThread	< Plan, Stack >
Plan	Array of Elements
PlanRule	< Condition, Goal, Plan >
BAction	PROLOG Term
Capability	< PreCondition , BAction , PosCondition >

Table 5.1: Summary of the Object Representations

5.3.1 Knowledge Bases

The knowledge bases store the agent configuration[3]. There are five knowledge bases depicted in Figure 5.2: (i) *belief base*; the (ii) *desire base*; the (iii) *intention base*; the (iv) *plan rules base*, and; the (v) *capability base*. These structures are detailed next.

Belief Base

The (i) *belief base* stores the set of closed formulas that represent the current situation. The definition of *belief* and *belief base* are similar to the one proposed in 2APL. A $\langle belief\rangle$ is a PROLOG expression and the belief base is a PROLOG knowledge base. The syntax for the initial belief base starts with the keyword "Beliefs :" followed by one or more $\langle belief\rangle$, as presented in the example below.

```
Beliefs:
pos(1,1).
base(g1,5,5).
```

Desire Base

The (ii) *desire base* implements the knowledge base for the desires. A major difference between 2APL and the proposed model is that 2APL lacks the definition of *desire*. In order to address this issue, I introduced the new element $\langle desire\rangle$ to the syntax. It is a term of the form:

$\langle desire\rangle$::="{" $\langle belquery\rangle$ "}" $[+ \mid -]$ $\langle goal\rangle$

Where $\langle goal\rangle$ expression is a conjunction of ground atoms; the flag "$[+ \mid -]$" is the *redo* flag; "+" means "$redo = true$", "-" or absent means "$redo = false$"; the $\langle belquery\rangle$ is the set of conditions to be tested.

The operation works as follows: the *context observer* monitors the entries in $\langle belquery\rangle$ for the desires that are not being processed yet. If the test action evaluates as true, then it sends an event to the *planning module* to generate a new intention.

Desires are represented as objects with the attributes:

$\langle Condition, Goal, redo\rangle$

[3]*Agent configuration*: I recall from Definition 14 that "the agent configuration is the tuple $\langle \mathcal{B}, \mathcal{D}, \mathcal{I}, \mathcal{P}\rangle$, where $\mathcal{B}$ is the *belief base*, $\mathcal{D}$ is the *desire base*, $\mathcal{I}$ is the *intention base*, and $\mathcal{P}$ is the *plan base*.

Where *Condition* is a set of PROLOG terms; *Goal* is a PROLOG term, and; *redo* is a *boolean*.

I argue that the inclusion of desires in the agent model does not influence the 2APL operational semantics because this structure is controlled by the context observer module. The syntax of the initial *desire base* starts with the keyword "Desires :" followed by one or more $\langle desire \rangle$, as in the example below.

```
Desires:
{pos(X,Y) AND NOT base(G,X,Y)} goBase(G).
```

Intention Base

The (iii) *intention base* contains the intentions that the agent is deliberating at a given moment. *Intentions* are control structures that are represented as objects with the following attributes:

$$\langle RunStatus, Condition, Goal, PlanningThread \rangle$$

Where: *RunStatus* is an integer that represents the current state, that is (0 for "paused", 1 for "run", and 2 for "review"); *Condition* is a set of PROLOG terms with the variables resolved during the intention processing; *Goal* is a PROLOG term, and; *PlanningThread* is an object with the following attributes:

$$\langle Plan, Stack \rangle$$

Where *Plan* contains the current plan, which is a sequence of actions, and *Stack* is the unification stack.

The deliberation cycle *animates* the agent by processing the planning threads for the intentions in "run" state. This functionality is implemented by the (vii) *planning module*, which is presented in Section 5.3.3.

Plan Rules Base

The (iv) *plan rules base* is an object base that stores the three types of practical reasoning rules defined in 2APL: planning goal rules (PG-Rules), procedure call rules (PC-Rules), and plan repair rules (PR-Rules). These elements are detailed below.

Planning Goal Rules. The (vii) *planning module* uses the planning goal rules (PG-Rules) to generate plans for given goals and situations. The rule ⟨*pgrule*⟩ is of the form:

$\langle goalquery \rangle \leftarrow \langle belquery \rangle \mid \langle plan \rangle;$

Where ⟨*goalquery*⟩ is the "head" of the rule, which describes the goal to be achieved if this planning rule is applied; ⟨*belquery*⟩ is the "guard", which contains the conditions where this rule can be applied, and; ⟨*plan*⟩ is the "body", which contains the sequence of actions to be executed. *Plan rules* are represented as objects with the following attributes:

$\langle Condition, Goal, Plan \rangle$

Where *Condition* is a set of PROLOG terms; *Goal* is a PROLOG term, and; *Plan* is a sequence of actions (i.e. an array of elements). Each action in *Plan* can be either a basic action or an abstract plan (i.e. a sub-goal).

The syntax of the *plan rules base* starts with the keyword "PG-rules:" followed by one of more ⟨*pgrule*⟩ elements, as presented in the example below:

```
PG-rules:
goBase(G) <- pos(X,Y) & base(G,Xb,Yb) & Yb > Y |
 {@mobileworld(North(),_);GoNorth();goBase(G)}.
goBase(G) <- pos(X,Y) & base(G,Xb,Yb) & Yb < Y |
 {@mobileworld(South(),_);GoSouth();goBase(G)}.
goBase(G) <- pos(X,Y) & base(G,Xb,Yb) & Xb < X |
 {@mobileworld(West(),_);GoWest();goBase(G)}.
goBase(G) <- pos(X,Y) & base(G,Xb,Yb) & Xb > X |
 {@mobileworld(East(),_);GoEast();goBase(G)}.
```

Procedure Call Rules. The *procedure call rules* is activated in response to messages received from other agents, events generated by the external environment, or the execution of abstract actions. The rule ⟨*pcrule*⟩ is of the form:

$\langle atom \rangle \leftarrow \langle belquery \rangle \mid \langle plan \rangle;$

Where ⟨*atom*⟩ is the "head" of the rule, which describes the event that triggers the rule (i.e. a message, an event, or an abstract action); ⟨*belquery*⟩ is the "guard", which contains the conditions where this rule can be applied, and ⟨*plan*⟩ is the "body", which contains the sequence of actions to be executed.

These rules are stored as *PlanRule* objects, such as for the PG-rules (described above). The syntax of the *procedure call rules base* starts with the keyword "PC-rules:" followed by one of more ⟨*pcrule*⟩ elements, as in the example below.

```
PC-rules:
message(A,inform,La,On,goldAt(X2,Y2)) <- not carry(gold) |
 {[ pos(X1,Y1)?; goTo(X1,Y1,X2,Y2);
 @blockworld(pickup(),_); PickUp();
 goTo(X2,Y2,3,3);
 @blockworld(drop(),_); AddGold()]

event(gold(X2,Y2),blockworld) <- not carry(gold) |
 {[ pos(X1,Y1)?; goTo(X1,Y1,X2,Y2);
 @blockworld(pickup(),_); PickUp();
 goTo(X2,Y2,3,3);
 @blockworld(drop(),_); AddGold()]

goTo(X1,Y1,X2,Y2) <- X1 < X2 |
 {[ @blockworld(east(),_); ChgPos(X1+1,Y1);
 goTo(X1+1,Y1,X2,Y2)]
```

This structure extends the proposed model and provides a practical solution for external event manipulation. They are operationalised by programming constructs in the deliberation cycle without compromising the operational semantics of the model.

Plan Repair Rules. The *plan repair rules* ⟨*prrule*⟩ has the following form:

$$\langle planvar \rangle \leftarrow \langle belquery \rangle \mid \langle planvar \rangle$$

This rule indicates that the current plan can be replaced by ⟨*planvar*⟩. The rules are processed during the plan revision stage in the deliberation cycle. *Plan repair rules* are represented as objects with the following attributes:

$$\langle Condition, Plan, NewPlan \rangle$$

Where *Condition* is a set of PROLOG terms; *Plan* and *NewPlan* are plans. The syntax of the *plan repair rules* starts with the keyword "PR-rules:" followed by one or more ⟨*prrule*⟩ elements, as in the example below:

```
PR-rules:
@mobileworld(East(),_);@mobileworld(East(),_);X <- true |
 {@mobileworld(North(),_);@mobileworld(East(),_);
@mobileworld(East(),_);@mobileworld(south(),_);X}
```

The deliberation cycle selects those entries whose "head" element matches the intended goal and the "guard" element holds with the belief base. The (vii) *planning module* implements this operation.

Capability Base

The (iv) *capability base* is an object base that stores the capabilities of the agent. The 2APL provides the syntax for specific actions such as belief update, communication, external interaction, and test action. The proposed model can accommodate these elements as described below.

Belief Update Actions. The *belief update actions* change the contents of the belief base when executed. These actions are defined in terms of pre- and post-conditions, and are related to the concept of *capability* introduced in Section 3.3.3. *Capabilities* are represented as objects with the attributes:

$$\langle PreCondition, BAction, PosCondition\rangle$$

Where $PreCondition$ is an instance of $Condition$, which is a set of PROLOG terms; $BAction$ is a PROLOG term that represents a basic action, and; $PosCondition$ is also an instance of $Condition$.

The syntax for the *capability base* starts with the keyword "BeliefUpdates :" followed by one or more belief update actions $\langle BelUpSpec\rangle$, whose syntax is:

$$(\text{“\{”}\langle belquery\rangle\text{“\}”}\ \langle beliefupdate\rangle\ \text{“\{”}\langle literals\rangle\text{“\}”})+$$

An example of capability base is presented below.

```
 BeliefUpdates:
 {pos(X,Y)} GoNorth() {NOT pos(X,Y), pos(X,Y+1)}
 {pos(X,Y)} GoSouth() {NOT pos(X,Y), pos(X,Y-1)}
 {pos(X,Y)} GoEast() {NOT pos(X,Y), pos(X+1,Y)}
 {pos(X,Y)} GoWest() {NOT pos(X,Y), pos(X-1,Y)}
```

The deliberation process selects the capability by looking for entries whose *basic action* matches the intended action and the *pre-condition* (i.e. $\langle belquery\rangle$) holds with the belief base.

Communication Actions. The *communication actions* allow inter-agent communication. These operations are implemented as programming constructs and extents the proposed model, providing a practical solution. The communication action $\langle sendaction \rangle$ is an expression of the form:

$$send(Receiver, Performative, Language, Ontology, Content)$$

Where *Receiver* is a name of the destination agent; $Performative$ is a speech act name (e.g. inform, or request); *Language* is the name of the language used to express the content of the message; *Ontology* is the name of the ontology used to give a meaning to the symbols in the content expression, and; $Content$ contains the content of the message.

The current 2APL platform is built on top of the FIPA-compliant JADE platform (see Bellifemine et al. [2005]), which provides the infrastructure for the communication process. It is possible to accommodate these specification using programming constructs with no side effect to the model's operational semantics.

External Actions. These actions change the external environment. The environment determines the effects of external actions, which may not be known to the agents during execution. In 2APL, an external action $\langle externalaction \rangle$ is an expression of the form:

$$@Env(ActionName, Return)$$

Where *Env* is the name of the environment (defined as a Java class in current 2APL implementation); $ActionName$ is a method call (of the Java class) that specifies the effect of the external action upon the environment; and *Return* is a list of values returned by the corresponding method.

I introduce a practical solution for this functionality *via* programming constructs in Section 5.4

Abstract Actions. The idea behind *abstract actions* is similar to a procedure call in imperative programming languages. An abstract action $\langle abstractaction \rangle$ is an expression of the form $\langle atom \rangle$, i.e. a first order expression in which the predicate starts with a lowercase letter. The execution of an abstract action passes parameters from the plan in which it occurs to another plan associated with it by a PC-rule. Programming constructs implement these actions in the prototype.

Test Action. The *test actions* allow the program to check whether the agent has certain beliefs and goals. Expressions of the form $\langle test \rangle$ consistof either a belief or a goal query expression, and can be used inside a plan to (i) instantiate variables in subsequent actions (if the test succeeds), or (ii) block the plan's execution if the test fails.

A *belief query* has the form $B(\phi)$, where ϕ consists of literals composed by conjunction and disjunction operators. A PROLOG query to the belief base implements this functionality, which returns the substitution set for the free variables.

A *goal query* has the form $G(\varphi)$, where φ consists of atoms composed by conjunction and disjunction operators, and is used to query to individual goals in the *intention base.*

Goal Dynamics Action. The action $\langle adoptgoal \rangle$, to adopt a goal, has two different forms:

- $adopta(\varphi)$ to add a goal φ to the beginning of the goal base, i.e. with higher priority in the execution queue, and;
- $adoptz(\varphi)$ to goal φ to the end of the goal base, i.e. with lower priority in the execution queue.

In addition, the action $\langle dropgoal \rangle$, to drop an existing goal, has three possible forms:

- $dropGoal(\varphi)$ to drop all goals that are a logical subgoal of ϕ;
- $dropSubGoal(\varphi)$ to drop all goals that have φ as a logical subgoal, and;
- $dropExactGoal(\varphi)$ to drop only the goal φ.

The application implements these actions *via* programming constructs. The action to "adopt a goal" is actually implemented as "adopt a desire". Similarly, the action to "drop a goal" is implemented as "drop a desire". Hence, the parameter φ is replaced by the object *Desire*, described before. This allows external modules to either assert or retract desires at run-time.

Summary

The 2APL language provides a clear and simple syntactic representation of the basic elements of the BDI architecture – i.e. beliefs, goals and plans, and describes the syntax of the elements in EBNF format. I conclude that the 2APL language syntax can be re-used largely unchanged, except for the insertion of the ⟨*desire*⟩ term. This specification is useful for defining the *parser module.*

Moreover, the 2APL specifications introduce practical solutions for features that were not addressed by the proposed model; namely inter-agent communication, executing external actions, and the processing of external events. These solutions add value to the proposed model.

Figure 5.3 presents the summary of the EBNF specifications. I highlight that the only differences in this specification are the replacement of the entry "Goal:" with "Desire:", and the introduction of the expression ⟨*desire*⟩. The other elements remain unchanged.

5.3.2 External Interfaces

The architecture also provides the interface to integrate external components; that is (ix) *actuators*, (x) *sensors*, and (xi) *parser.*

These elements are implemented via programming constructs that provide two interfaces: *an external interface* that supports interaction with field technologies, and *an internal interface* for interaction between the agent's structures.

The segregation between core and external interfaces allows the former to remain unchanged when new technology enablers are "plugged in" to the latter. This design promotes *flexibility*, *compatibility*, and *stability* of the application.

Actuators

The (ix) *actuator module* implements the interface to field technologies to execute physical actions. For example, the actuators display notifications on the device's screen.

The executions of actuator operations can fail, resulting what is called as *physical exceptions.* Programming constructs capture these exceptions, which fire events to the (viii) *learning module.*

Moreover, the agent knows what to expect from the execution of a physical action. This is specified in the *pos-condition* element. Hence, the module verifies whether the pos-condition holds with the belief base; if not, then this situation is classified as a *logical exception.* These events are processed as above.

Sensors

The (x) *sensor module* implements the interface to the field technologies that perceive the environment and update the belief base. *Sensors* interface with

```
2APLM_Prog         :== ("Include:" ⟨ident⟩
                    |   "BeliefUpdates:" ⟨BelUpSpec⟩
                    |   "Beliefs:" ⟨belief⟩
                    |   "Desires:" ⟨desires⟩
                    |   "Plans:" ⟨plans⟩
                    |   "PG-rules:" ⟨pgrules⟩
                    |   "PC-rules:" ⟨pcrules⟩
                    |   "PR-rules:" ⟨prrules⟩)*
⟨BelUpSpec⟩        ::= ( "{" ⟨belquery⟩ "}" ⟨beliefupdate⟩ "{" ⟨literals⟩ "}" )+
⟨belief⟩           ::= ( ⟨ground_atom⟩ "." | ⟨atom⟩ ":-" ⟨literals⟩ "." )+
⟨desires⟩          ::= ⟨desire⟩(","⟨desire⟩)*
⟨desire⟩           ::= "{" ⟨belquery⟩ "}" [+ | −] ⟨goal⟩
⟨goal⟩             ::= ⟨ground_atom⟩ ("and" ⟨ground_atom⟩)*
⟨baction⟩          ::= "skip" | ⟨beliefupdate⟩ | ⟨sendaction⟩ | ⟨externalaction⟩
                    |   ⟨abstractaction⟩ | ⟨test⟩ | ⟨adoptgoal⟩ | ⟨dropgoal⟩
⟨plans⟩            ::= ⟨plan⟩("," ⟨plan⟩)*
⟨plan⟩             ::= ⟨baction⟩ | ⟨sequenceplan⟩ | ⟨ifplan⟩ | ⟨whileplan⟩ |
                        ⟨atomicplan⟩
⟨beliefupdate⟩     ::= ⟨Atom⟩
⟨sendaction⟩       ::= "send(" ⟨iv⟩ "," ⟨iv⟩ "," ⟨atom⟩ ")"
                    |   "send(" ⟨iv⟩ "," ⟨iv⟩ "," ⟨iv⟩ "," ⟨iv⟩ "," ⟨atom⟩ ")"
⟨externalaction⟩   ::= "@" ⟨ident⟩ "(" ⟨atom⟩ "," ⟨Var⟩ ")"
⟨abstractaction⟩   ::= ⟨atom⟩
⟨test⟩             ::= "B(" ⟨belquery⟩ ")" | "G(" ⟨goalquery⟩ ")" | ⟨test⟩ "&"
                        ⟨test⟩
⟨adoptgoal⟩        ::= "adopta(" ⟨goalvar⟩ ")" | 'adoptz(" ⟨goalvar⟩ ")"
⟨dropgoal⟩         ::= "dropgoal(" ⟨goalvar⟩ ")" | "dropSubgoal(" ⟨goalvar⟩ ")"
                    |   "dropExactgoal(" ⟨goalvar⟩ ")"
⟨sequenceplan⟩     ::= ⟨plan⟩ ";" ⟨plan⟩
⟨ifplan⟩           ::= "if" ⟨test⟩ "then" ⟨scopeplan⟩ ("else" ⟨scopeplan⟩)?
⟨whileplan⟩        ::= "while" ⟨test⟩ "do" ⟨scopeplan⟩
⟨atomicplan⟩       ::= "[" ⟨plan⟩ "]"
⟨scopeplan⟩        ::= "{" ⟨plan⟩ "}"
⟨pgrules⟩          ::= ⟨pgrule⟩+
⟨pgrule⟩           ::= ⟨goalquery⟩? "←" ⟨belquery⟩ "|" ⟨plan⟩
⟨pcrules⟩          ::= ⟨pcrule⟩+
⟨pcrule⟩           ::= ⟨atom⟩ "←" ⟨belquery⟩ "|" ⟨plan⟩
⟨prrules⟩          ::= ⟨prrule⟩+
⟨prrule⟩           ::= ⟨planvar⟩ "←" ⟨belquery⟩ "|" ⟨planvar⟩
⟨goalvar⟩          ::= ⟨atom⟩("and" ⟨atom⟩)*
⟨planvar⟩          ::= ⟨plan⟩ | ⟨Var⟩ | "if" ⟨test⟩ "then" ⟨scopeplanvar⟩ ("else"
                        ⟨scopeplanvar⟩)? | "while" ⟨test⟩ "do" ⟨scopeplanvar⟩ |
                        ⟨planvar⟩ ";" ⟨planvar⟩
⟨scopeplanvar⟩     ::= "{" ⟨planvar⟩ "}"
⟨literals⟩         ::= ⟨literal⟩(","⟨literal⟩)*
⟨literal⟩          ::= ⟨atom⟩ | "not" ⟨atom⟩
⟨ground_literal⟩   ::= ⟨ground_atom⟩ | "not" ⟨ground_atom⟩
⟨belquery⟩         ::= "true" | ⟨belquery⟩ "and" ⟨belquery⟩ | ⟨belquery⟩ "or"
                        ⟨belquery⟩
                    |   "(" ⟨belquery⟩ ")" | ⟨literal⟩
⟨goalquery⟩        ::= "true" | ⟨goalquery⟩ "and" ⟨goalquery⟩ | ⟨goalquery⟩ "or"
                        ⟨goalquery⟩ | "(" ⟨goalquery⟩ ")" | ⟨atom⟩
⟨iv⟩               ::= ⟨ident⟩ | ⟨Var⟩
```

Figure 5.3: Summary of the 2APL-M EBNF syntax

field technologies that collect data; for example, a sensor for location can access a location-based system to collect the current location, which then updates the current position representation in the belief base.

Parser

Finally, the (xi) *parser module* loads the *agent program* from a file and initialises the knowledge bases. Executed at initialisation, The module contains the interface to the field technology that provides access to physical files. Once the file is open, the process "parses" the textual content following the EBNF syntax rules.

5.3.3 Deliberation Cycle

The *deliberation cycle* animates the agent. It implements the operations to transition the state of the *agent configuration.*

For the sake of modularity, I propose to distinguish the programming constructs in three groups, following the same structure introduced in the previous chapter: the (vi) *context observer*, the (vii) *planning module*, and the (viii) *learning module.*

Context Observer

The *context observer* implements the programming structures to evaluate events associated with belief base updates and to coordinate the planning module. It implements the methods to:

- capture the events;
- filter the relevant changes, that is, it implements the functionality of the meta-operator $\mathsf{impact}(\beta, \varphi, V_r)$;
- process events associated with desire base and intention base updates, including the *function to extract relevance filters.*
- dispatch events to the planning process.

I recall that the function to calculate proximity conditions implements an important functionality in this model. In short, the process calculates proximity situations based on the "guard" of indexed elements. A practical solution for its functionality is presented in Section 5.4.

Planning Module

The *planning module* implements the operations to form plans to achieve goals. The processing is trigged by an event to “start to deliberate on goal φ” from the (vi) *content observer*. The processing involves several steps, as described in Section 3.4:

- generate a new intention in the intention base;
- initialise a new planning thread for this intention;
- depending on the prioritisation rules, set this intention to “run” mode;
- include this elements in the processing loop, and;
- processes each step of the plan in the processing loop, resolving the process for each intention at the time.

Learning Module

The *learning module* implements the programming structures to (i) analyse the cause of exceptions in basic action executions, and (ii) adjust operational values to improve the application’s performance.

In Section 4.3.3, I mentioned that these functions should be provided as *abstract interfaces* that can be overloaded by programming constructs in the implementation. This module provides the structures to:

- capture incoming events from the planning module;
- analyse the events; this is an abstract method that must be overloaded in the implementation with the logic that correlates events and the operational conditions;
- adjust the operation values; operationalised as events fired to the knowledge databases to update the current operational values; this is also an abstract method, and;
- dispatch events to the planning module requesting *plan revision*.

I stress once again, that issues of *what* and *how* to learn are out of the scope of this work; therefore, I do not detail their operational aspects.

5.3.4 Operations

In the remainder of the chapter, I detail the processes that compose the deliberation cycle. I relate these structures to the concepts introduced in the previous chapter.

Environmental Information Updates

The (x) *sensor module*, depicted in Figure 5.4, operates as a background loop. At one side, this module has an interface to the field technologies that "senses" changes in the environment; on the other, it has the programming constructs to interface with the core components, which will process the sensed facts accordingly.

This module is the motive force behind the agent's life cycle. That is, although belief base updates can result from the execution of basic actions (by the (vii) *planning module*) and internal adjustments (by the (viii) *learning module*) these operations can be understood as part of the processing loop; thus, they can be labelled as *mental operations.* It is indispensable to have sensors collecting information from the external environment to trigger the conditions that enable the deliberation of new intentions (from existing desires).

The sensor updates the belief base, which triggers the (d) *function to process belief base updates.* This function processes the event $\langle update, \beta, \mathcal{B}\rangle$ as described below[4]:

- first, it *verifies* whether the belief β is being observed by any entry in the *relevance filter base*; I recall that a belief base entry is a PROLOG term, and that entries in the relevance filter base have the form $\langle \beta, E\rangle$;
- if the above is true, then for every $\langle \beta', E'\rangle$ whose β' matches β, it does the following:
  - if the entry E' is a desire[5] and there is no in-processing intention for this desire, then it *calculates the proximity condition*; if it returns an "in" condition, then the process fires the event to start the processing;

[4] *Operational Semantics*: I relate these operations to the operational semantic for the meta-operator $\mathsf{impact}(\beta, \varphi, V_r)$.

[5] *Type of the entry*: the practical solution to assert the class of an entry is provided by the programming language; for example, the Java language has the operation "E instanceof Class" for this purpose.

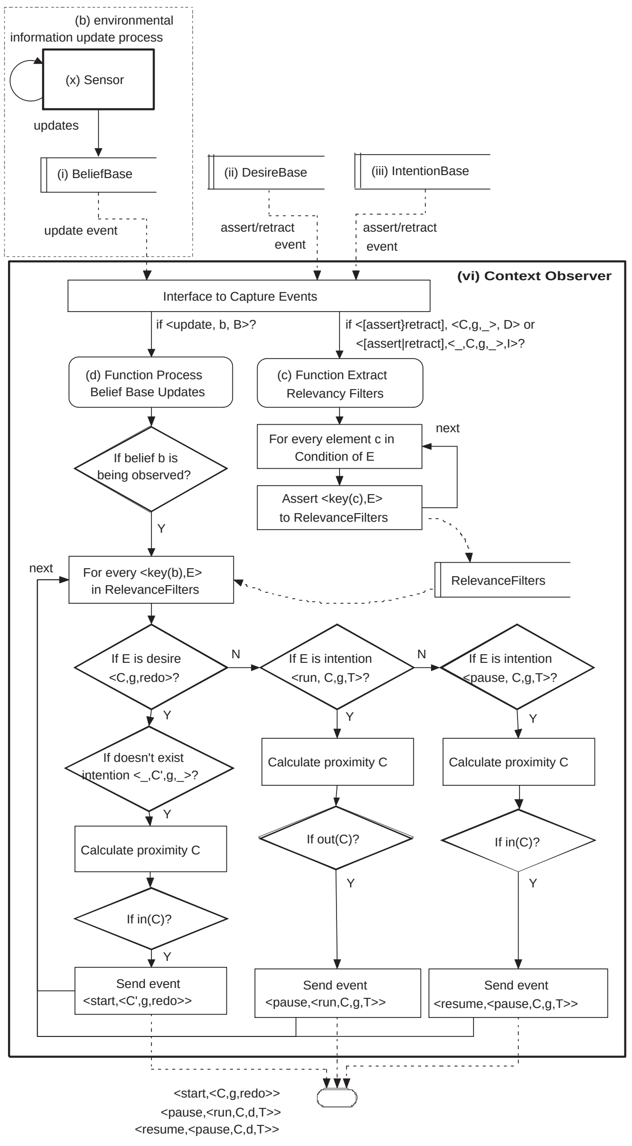

(b) environmental information update process
(x) Sensor
updates
(i) BeliefBase
update event
(ii) DesireBase
assert/retract event
(iii) IntentionBase
assert/retract event
(vi) Context Observer
Interface to Capture Events
if <update, b, B>?
if <[assert}retract], <C,g,_>, D> or <[assert|retract],<_,C,g,_>,I>?
(d) Function Process Belief Base Updates
(c) Function Extract Relevancy Filters
For every element c in Condition of E
next
Assert <key(c),E> to RelevanceFilters
If belief b is being observed?
Y
RelevanceFilters
next
For every <key(b),E> in RelevanceFilters
If E is desire <C,g,redo>?
N
If E is intention <run, C,g,T>?
N
If E is intention <pause, C,g,T>?
Y
If doesn't exist intention <_,C',g,_>?
Calculate proximity C
If out(C)?
If in(C)?
Send event <start,<C',g,redo>>
Send event <pause,<run,C,g,T>>
Send event <resume,<pause,C,g,T>>
<start,<C,g,redo>>
<pause,<run,C,d,T>>
<resume,<pause,C,d,T>>

Figure 5.4: Workflow of the Context Observer

- if the entry E is an intention with "run" status, then it calculates the proximity condition; if it returns the "out" condition, then it fires the event to pause the processing;
- otherwise, if the entry E is an intention with "paused" status, then it calculates the proximity condition; if it returns an "in" condition, then it fires an event to resume the processing.

I note the decision to fire the *resume* event without checking for possible conflicts was taken to preserve the modularity of the design. The conflict and prioritisation rules are observed by the (vii) *planning module*.

This design also implies that paused intentions are continuously re-evaluated. This verification must be implemented as a loop somewhere in the module, and embedded in the context observation process as an opportunistic approach. Therefore, this design decision does not compromise overall application performance.

Processing Start

The operation continues as follows. The events $\langle start, d\rangle$, where $d = \langle C, \varphi, redo\rangle$, are captured by the (vii) *planning module*. They trigger the (e) *function for intention deliberation* depicted in Figure 5.5, which creates a new $Intention$ object with the $PlanThread$ objects initialised accordingly. These steps are detailed below.

- the function *verifies* whether another in-processing intention conflicts with the goal φ. As the goals are represented as PROLOG terms, the practical solution is to look up entries in the *intention base* whose *Goal* element implies entries $conflict(Goal, \varphi)$ in the *belief base*;

- if none, then the process *creates* a new $Intention$ object with the attributes $\langle run, C, d, \emptyset\rangle$ and asserts it to the (iii) *intention base*;

- otherwise, if there is a matching conflict rule, then the process *verifies* which goal has a higher priority; again, the practical solution is to look for the entries $priority(\varphi, N_\varphi)$ and $priority(\varphi', N_{\varphi'})$ in the belief base and compare the values of N_φ and $N_{\varphi'}$. If $N_\varphi > N_{\varphi'}$ – that is the in-processing element has a higher priority –, then the process creates a new $Intention$ object with the attributes $\langle pause, C, d, \emptyset\rangle$; otherwise, it creates a new intention with the attributes $\langle run, C, d, \emptyset\rangle$ *and* updates the status flag to pause the processing of φ.

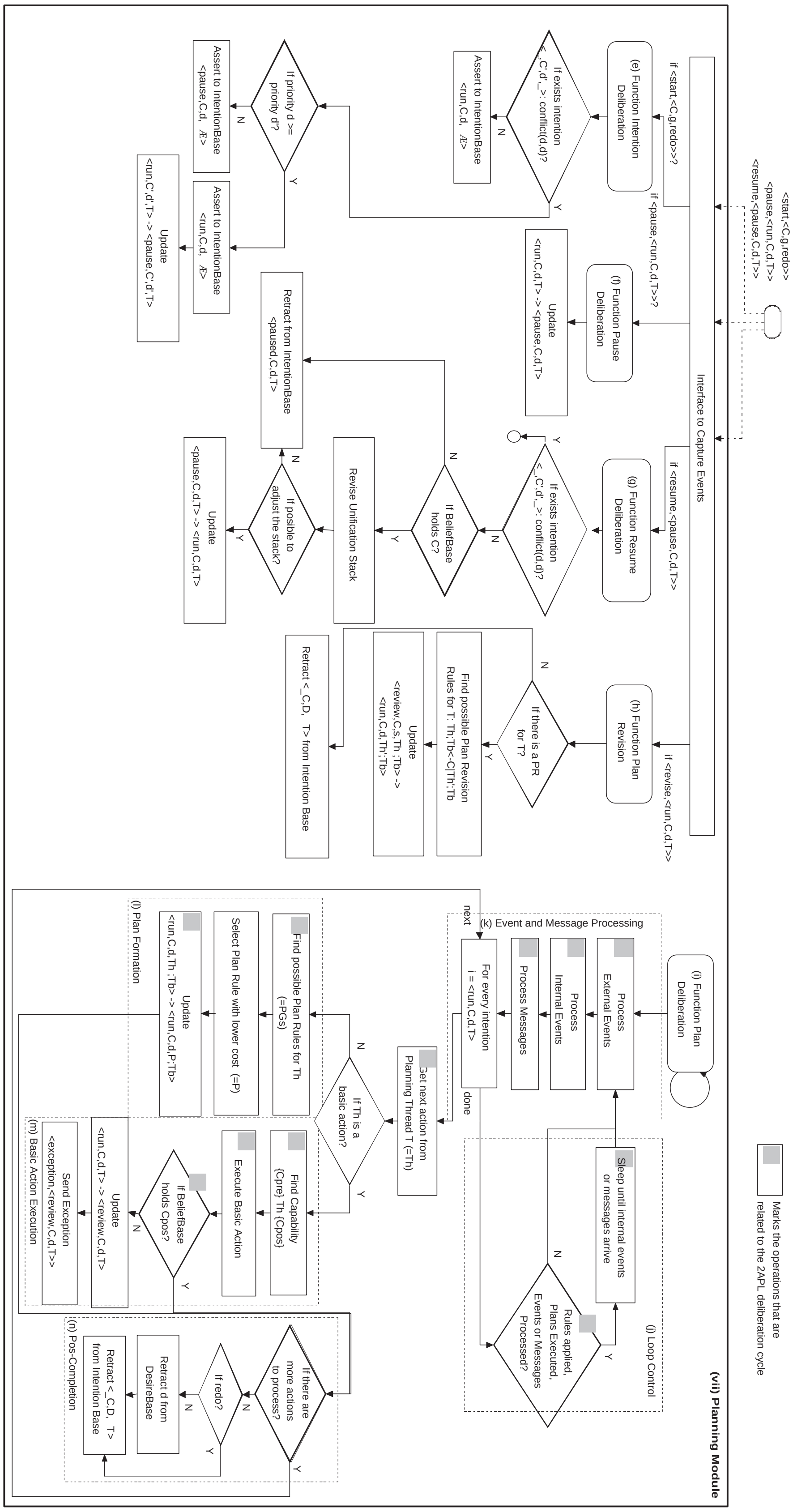

(vii) Planning Module
Marks the operations that are related to the 2APL deliberation cycle
<start,<C,g,redo>>
<pause,<run,C,d,T>>
<resume,<pause,C,d,T>>
Interface to Capture Events
if <start,<C,g,redo>>?
(e) Function Intention Deliberation
If exists intention <_,C',d',_>: conflict(d,d')?
Assert to IntentionBase <run,C,d, ∅>
If priority d >= priority d'?
Assert to IntentionBase <pause,C,d, ∅>
Assert to IntentionBase <run,C,d, ∅>
Update <run,C',d',T> -> <pause,C',d',T>
if <pause,<run,C,d,T>>?
(f) Function Pause Deliberation
Update <run,C,d,T> -> <pause,C,d,T>
if <resume,<pause,C,d,T>>
(g) Function Resume Deliberation
If exists intention <_,C',d',_>: conflict(d,d')?
If BeliefBase holds C?
Retract from IntentionBase <paused,C,d,T>
Revise Unification Stack
If posible to adjust the stack?
Update <pause,C,d,T> -> <run,C,d,T>
if <revise,<run,C,d,T>>
(h) Function Plan Revision
If there is a PR for T?
Find possible Plan Revision Rules for T: Th,Tb<-C|Th';Tb
Update <review,C,s,Th,;Tb> -> <run,C,d,Th';Tb>
Retract <_,C,D, T> from Intention Base
(i) Function Plan Deliberation
(j) Loop Control
Sleep until internal events or messages arrive
Rules applied, Plans Executed, Events or Messages Processed?
(k) Event and Message Processing
Process External Events
Process Internal Events
Process Messages
For every intention i = <run,C,d,T>
next
done
Get next action from Planning Thread T (=Th)
If Th is a basic action?
(l) Plan Formation
Find possible Plan Rules for Th (=PGs)
Select Plan Rule with lower cost (=P)
Update <run,C,d,Th ;Tb> -> <run,C,d,P;Tb>
(m) Basic Action Execution
Find Capability {Cpre} Th {Cpos}
Execute Basic Action
If BeliefBase holds Cpos?
Update <run,C,d,T> -> <review,C,d,T>
Send Exception <exception,<review,C,d,T>>
(n) Pos-Completion
If there are more actions to process?
If redo?
Retract d from DesireBase
Retract <_,C,D, T> from Intention Base
Y
N

Figure 5.5: Workflow of the Planning Module

I also note that the assertion of new intentions generates update events, which are fed back to the (vi) *context observer* and will, ultimately, update the relevance filters. At this point, the conditions for the new intentions are also being observed by the context observer module.

Pausing the Execution

The events $\langle pause, d\rangle$, where $d = \langle C, \varphi, redo\rangle$, are also captured by the (vii) *planning module.* They trigger the (f) *function to pause execution* depicted in Figure 5.6.

The processing is simple: the routine looks for objects whose *Goal* attribute matches φ' and switches its status flag to "paused". The processing loop design ensures that switching the status flag causes no side effects.

I highlight the consequences of this action upon the operations of the (vi) *context observer.* The relevance filters are still observing the (just) paused intention, hence, while the condition is evaluated as "in" the window of opportunity, the process keeps sending events to resume the processing. The (vii) *planning module* receives these events, however the processing itself does not resume whilst the conflicting situation persists.

There are two situations when this verification loop will stop: (i) when the conflicting intention completes its processing and is retracted from the intention base (see the diagram for the (j) *processing loop*), or (ii) as the environment changes, in which case the condition now evaluates as "out" of the window of opportunity.

This design provides a consistent solution for the process of continuously verifying the reconsideration conditions.

Resuming the Execution

The third set of events from the context observer are the events $\langle resume, d\rangle$ from the (g) *function to resume execution.* This process involves a series of practical solutions, as introduced below:

- the process *verifies* whether the paused processing thread conflicts with any other in-processing execution;
- if there are no conflicting elements, then the process *verifies* whether the *Condition* element is still valid. If the condition is no longer valid, then it *retracts* the intention, and;
- finally, it implements the functionality for the meta-operator $\mathsf{revise}(\Theta, \mathcal{B}, \Theta')$ to revise the contents of the unification stack Θ with regards to the cur-

rent situation $\mathcal{B}$ resulting Θ' with revised values or $\emptyset$ if the revision is not possible.

There are some situations where the execution cannot be resumed, as demonstrated in this example.

Example 9. (situation when the execution cannot be resumed)

> *For example, let us consider that the agent commits to go to a base located at position x. Then, while moving towards that base, a new base is asserted at position y and with a higher priority than x. The execution leads to a goal switch at position z. The processing of $go(x)$ is paused and the unification stack holds the information that the current location is z. The agent starts to deliberate $g(y)$. After the agent completes $go(y)$, the event to* resume *$go(x)$ is received by the planning process. However, the condition $pos(z)$ no longer holds with the environment, as the agent is at y.*

If the process verifies that the condition is still valid, then it can try to *adjust the unification stack* to resume the execution. Otherwise, the solution is to drop the processing thread and let the context observer re-process the desire – i.e. start from scratch.

The process to functionality of $\mathsf{revise}(\Theta, \mathcal{B}, \Theta')$ is implemented via a programming construct, as explained below.

1. unification marks exist in the unification stack as tuples $\langle atom, variable, value\rangle$; for example, if the unification stack contains the entries:

   $\langle pos(X,Y), X, 5\rangle$
   $\langle pos(X,Y), Y, 2\rangle$

   it means that the variable X of the condition element $pos(X,Y)$ unified with the belief base with value $X = 5$, and $\langle pos(X,Y), Y, 2\rangle$ means that the variable Y of the condition element $pos(X,Y)$ unified with the belief base with value $Y = 2$;

2. then, it is possible to re-process these unifications by adjusting the *value* attribute in these tuples with the new values from the belief base; for example, if the belief base now contains $pos(8,3)$, then the adjustment process replaces the attributes accordingly, making the new entries in the unification stack:

$\langle pos(X, Y), X, 8\rangle$
$\langle pos(X, Y), Y, 3\rangle$

Of course, it requires that the elements of the unification stack are accessible via a programming interface. The PROLOG engine used for the implementation of the prototype – the PROLOG-M, as described below – supports this requirement.

I highlight one shortcoming in this approach: the process does not consider the updates of priority parameters at run time. That is, if the priorities are updated so that the paused intention assumes a higher priority than the in-processing one, the process keeps the paused intention. This issue can be remedied if the application implements a practical solution that analyses the status of the elements during the update process and updates them accordingly.

In addition, keeping paused processing threads uses more memory resources. I argue that this is an acceptable shortcoming as (i) there is a growing availability of memory resources in current mobile devices, and (ii) the reduction of processor usage implies saving battery, which is a major constraint in mobile computing.

Revising Failed Plans

The (h) *function for plan revision* processes the events from the learning module to
$\langle revise, \langle St, C, d, T\rangle\rangle$. This process involves the following steps:

- it *looks up* all possible PR-Rule's $\langle pr_{cond}, pr_{head}, pr_{new}\rangle$ whose pr_{head} matches the current plan p' in the $PlanningThread$;
- from the set above, it *selects* those rules whose pr_{cond} holds with the belief base;
- it *selects* the first element and replace p' with pr_{new}, and;
- it updates the status from "review" to "run".

I note that the "review" status is introduced as a practical solution, marking the planning threads that resulted in *exceptions* during the basic action execution stage. These events are captured and processed by the (viii) *learning module*, which generates the "revise" events at the end of the processing.

If there are no PR-Rules available, then the process *retracts* the intention from the intention base. This practical solution is consistent with the idea that there is no alternative course of action if one of the actions cannot be execute.

One limitation of this model is that a planning thread in “review” mode cannot be paused.

Background Plan Execution

Finally, the (i) *function for plan execution* implements the core functionality of the planning module; that is, it forms plans to achieve intended goals by combining the elements from the knowledge bases. This function works as a background loop, which is initialised during the agent initialisation process.

The methods marked with *grey boxes* had the concept re-used from 2APL definitions. The other methods are practical solutions adopted to accommodate the new specifications. I introduce these modules below and detail their functionality in the next sub-section.

The operations can be segmented in five stages:

(j) Loop Control. This first stage is required to keep the (i) *function for plan execution* running as a background loop. This concept was re-used from the 2APL deliberation cycle.

The process verifies whether there are plans to execute (i.e. running intentions with pending actions), messages to process, or events to execute. If none, then it enters in a *sleep state*, which can be described as a background loop;

(k) Event and Message Processing. This stage processes internal and external events and messages prior to the next batch of plan executions. This concept was re-used from the 2APL deliberation cycle.

This structure is followed by a *for-loop* to process the next step of all processing threads in *run* state. This is an extension upon the 2APL design to accommodate simultaneous intention deliberation. The functionality is similar than that in 2APL as described below.

- process external events, which are handled by PC-Rules;
- process internal events, such as the events generated by the processing of PC-Rules (above), and;
- process queued messages, also handled by PC-Rules.

In resume, the method cross-relates the events in the queue with PC-Rules by matching the "head" element and the condition. If both hold, it processes the "body" element of that rule. This functionality is implemented by a programming construct whose operational semantics was not defined in the proposed model.

(l) Plan Execution. If the first element in the sequence that represent the current plan *is not a basic action*, then the process flows as follows:

- it *looks up* all possible PG-Rule $\langle pg_{cond}, pg_{goal}, pg_{body}\rangle$ whose pr_{goal} matches the current sub-goal g';
- it *calculates* the rule with lower cost[6], and;
- it replaces g' with pr_{body}.

(m) Basic Action Execution. If the element *is a basic action*, then the processing flows as follows:

- it *looks up* all capabilities $\langle cp_{precond}, cp_{bact}, cp_{posccond}\rangle$ whose cp_{bact} matches the current action a';
- from the set above, it selects the entries whose $cp_{precond}$ holds with the belief base;
- from the set of possible capabilities above, it *selects* the first entry (see disclaimer below);
- it *executes* the basic action; this functionality is implemented by a programming construct, described in Section 5.4;
- next, it verifies if $cp_{posccond}$ holds with the model new belief baseafter the basic action has been executed;
- if the above is false, then it *fires an event* reporting an exception to the (viii) *learning module.*

The action of selecting the first entry is a practical solution when multiple options are available.

[6] *Calculating costs*: This action implies that there is a method $cost(pg_{body})$ that calculates the cost of executing a PG-Rule in the current environment, which is implemented as a practical solution, described in Section 5.4.

(n) Pos-Completion. After the basic action has been executed, it is removed from the sequence of actions. The plan is completed when there are no further sub-goals to be processed. In this case, a *cleaning up* process takes place.

First, the process verifies whether the *redo* flag associated with the in-processing desire is false. If so, then it retracts the desire to avoid it re-processing, and retracts the (now) processed intention. Both operations update the knowledge bases that fire events to the (vi) *context observer*. These events signal that module to update its internal structures (i.e. the entries in the relevance base). The process loops back to the *for-loop control* to get the next intention to be processed. If there are no other intentions in "run" state, then the process goes back to the (i) *loop control*.

Agent Initialisation.

The agent life-cycle starts with the (a) *agent initialisation process*. The (xi) *parser module* loads the configuration from the program file and configures the elements creating the *initial agent state*. The steps are described below.

- the parser module *opens* the program file via the interface to the underlying field technology;
- it *parses* each line of the program file using the definitions described in the EBNF syntax;
- it *creates* the object representations based on this information, and;
- it *asserts* these objects to the knowledge bases.

This process generates events associated with knowledge base updates, which are captured by the (vi) *context observer*.

During initialisation, the events associated with belief base updates do not yet trigger the functions. This design decision was made because it is not possible to deliberate intentions whilst the knowledge bases are being populated; that is, the agent configuration is still incomplete. Rather, they are accumulated into the *initialisation queue* and processed during the agent starting process.

Learning from Exceptions

For the sake of this presentation, I am not detailing the internal operations of the (viii) *learning module*. I claim that the platform must provide the structures to let the implementation to replace the functionality with practical solutions.

As depicted in Figure 5.6, these operations are initiated by events $\langle exception, PT \rangle$ fired from the (vii) *planning module*. The learning module receives these events, analyses the causes, promotes internal adjustment, and fires the event $\langle revise, PT \rangle$ to the planning module.

In the meantime, the planning thread status is set to "review", which can be read as "*operation under review*". The *revise* event forces the planning module to process the PR-Rules upon the current plan.

Of course, only applying the PR-Rules can fix the situation. In this case, the (viii) *learning module* can be viewed as a proxy for the exception events.

5.3.5 Summary

Figure 5.6 provides the complete overview of the operations workflow.

I highlight that the deliberation cycle is designed to allow pausing the execution by just switching the status flag; in which case, the *for-loop control* does not select that intention for processing.

I also note that the processing is not actually happening in parallel. Instead, it is interleaving between each execution. This is due to a physical limitation inherited from one-processor systems where (physically) the processor can implement only one interaction per time. However, multi-processor systems can execute several planning threads in parallel, which leaves this option open for future works.

The agent starts its operations through a call to the method *start()* in the programming interface. This method implements the following steps:

- the application *processes* the entries in the initialisation update queue, which can result in the events that trigger intention deliberation processes if any of the updates configure the belief base (in which case, the event activates the (vii) *planning module*);

- it *initialises* the (x) *sensor module*, which runs a background loop that keeps observing (i.e. "sensing") the changes in the physical environment, and;

- it *initialises* the (h) *function for plan execution*, which also runs a background loop that process the next steps of the execution.

Next, I describe the practical solutions adopted in the implementation.

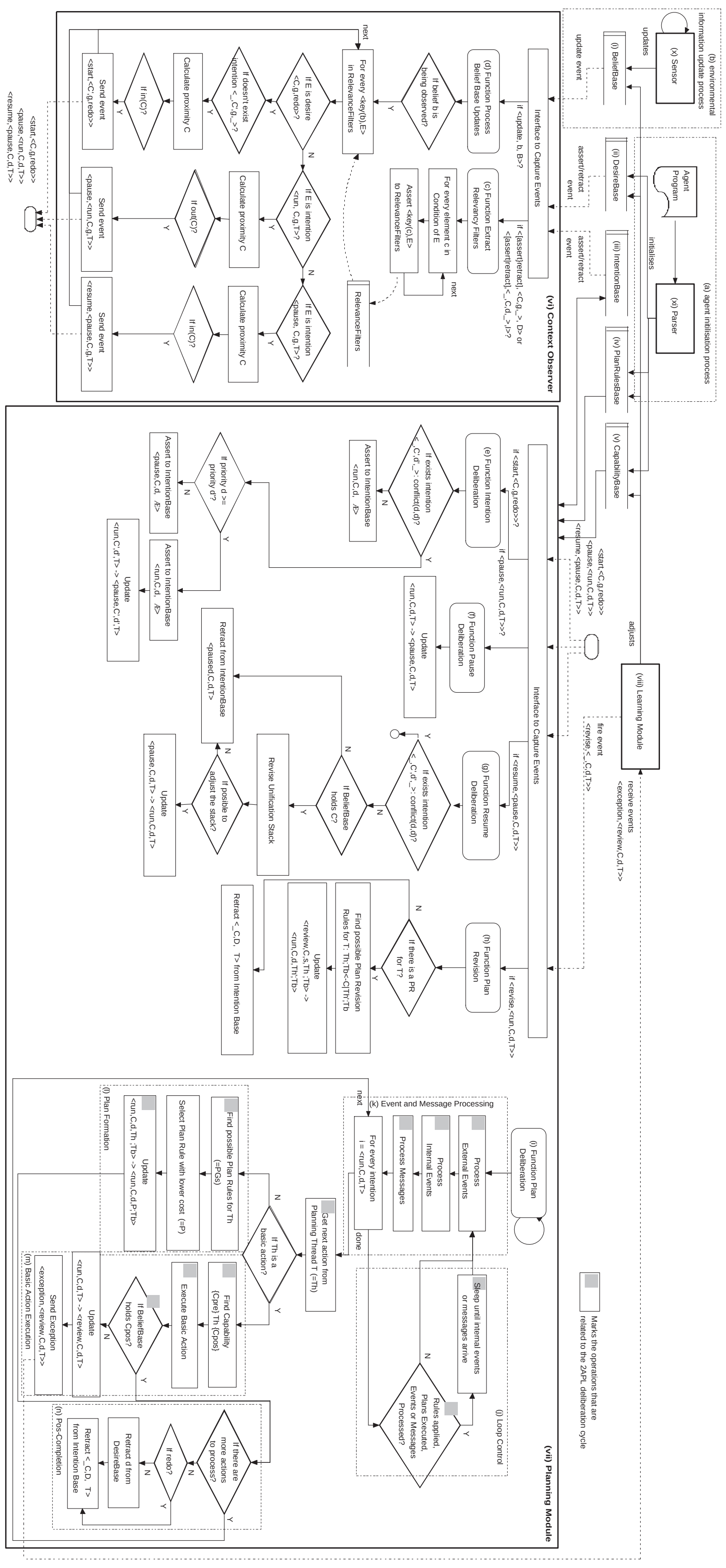

(b) environmental information update process
(x) Sensor
updates
(i) BeliefBase
update event
Agent Program
(a) agent initialisation process
initialises
(xi) Parser
(ii) DesireBase
assert/retract event
(iii) IntentionBase
(iv) PlanRulesBase
(v) CapabilityBase
adjusts
(viii) Learning Module
fire event <revise,<_,C,d,T>>
receive events <exception,<review,C,d,T>>
(vi) Context Observer
Interface to Capture Events
If <update, b, B>?
(d) Function Process Belief Base Updates
If belief b is being observed?
For every <key(b),E> in RelevanceFilters
If E is desire <C,g,redo>?
If doesn't exist intention <_,C,g,_>?
Calculate proximity C
If in(C)?
Send event <start,<C,g,redo>>
If E is intention <run, C,g,T>?
If out(C)?
Send event <pause,<run,C,g,T>>
If E is intention <pause, C,g,T>?
Send event <resume,<pause,C,g,T>>
(c) Function Extract Relevancy Filters
For every element c in Condition of E
Assert <key(c),E> to RelevanceFilters
RelevanceFilters
next
(vii) Planning Module
Interface to Capture Events
(e) Function Intention Deliberation
If exists intention <_,C',d',_> conflict(d,d')?
Assert to IntentionBase <run,C,d,_,A>
If priority d >= priority d'?
Assert to IntentionBase <pause,C,d,_,A>
Update <run,C',d',T> -> <pause,C',d',T>
(f) Function Pause Deliberation
Update <run,C,d,T> -> <pause,C,d,T>
(g) Function Resume Deliberation
If BeliefBase holds C?
Revise Unification Stack
If posible to adjust the stack?
Retract from IntentionBase <paused,C,d,T>
Update <pause,C,d,T> -> <run,C,d,T>
(h) Function Plan Revision
If there is a PR for T?
Find possible Plan Revision Rules for T: Th;Tb<-C|Th';Tb
Update <review,C,s,Th;;Tb> -> <run,C,d,Th';Tb>
Retract <_,C,D,_,T> from Intention Base
(i) Function Plan Deliberation
(k) Event and Message Processing
Process External Events
Process Internal Events
Process Messages
For every intention i = <run,C,d,T>
Get next action from Planning Thread T (=Th)
If Th is a basic action?
(l) Plan Formation
Find possible Plan Rules for Th (=PGs)
Select Plan Rule with lower cost (=P)
Update <run,C,d,Th;Tb> -> <run,C,d,P;Tb>
Find Capability {Cpre} Th {Cpos}
Execute Basic Action
If BeliefBase holds Cpos?
Update <run,C,d,T> -> <review,C,d,T>
(m) Basic Action Execution
Send Exception <exception,<review,C,d,T>>
(j) Loop Control
Sleep until internal events or messages arrive
Rules applied, Plans Executed, Events or Messages Processed?
If there are more actions to process?
If redo?
Retract d from DesireBase
(n) Pos-Completion
Retract <_,C,D,_,T> from Intention Base
Marks the operations that are related to the 2APL deliberation cycle

Figure 5.6: Complete Workflow

5.4 Practical Solutions

A number of practical solutions have been adopted to operationalise the implementation. These solutions cover the *gaps* in the specifications provided in Chapter 3.

I focus on the practical solutions that require detailed explanation due to their importance to the implementation: the functions to calculate proximity and the costs of plans, and the process to execute basic actions. These functions are presented below.

5.4.1 Calculating Proximity

The complexity of developing the function to calculate proximity is mitigated if one considers the requirements for a restricted knowledge domain. For example, in mobile services one can consider only time and space conditions.

I revisit the illustrative scenario proposed in Section 4.3.1. The agent has the desire to download the files for the upcoming meetings. Each condition represents the time of the appointment as the term $time(HHMM)$, where $HHMM$ is the string representation of the time in hours and minutes, the venue as the term $place(X,Y)$, where X and Y describe the physical coordinates, and possibly, another condition terms that describe specific requirements. Thus, the desire is represented as:

$$\langle\{meeting(M, time(HHMM), place(X,Y)), c^1, \ldots, c^n\},\\ downloadFiles(M), false\rangle$$

The number of files to be downloaded and the size of the window of opportunity (WoO) vary randomly per meeting. Let us say that for each meeting M there are these entries: $files(M,F)$ is the list of files associated with M; $woo(M,D,T)$ is the size of the window of opportunity associated to M, where D is the space and T is the time. The current position is represented as $place(X,Y)$, the current time is represented as $time(HHMM)$, and the agent contains the implementation to calculate physical distances as a Manhattan distance ($\Delta_x + \Delta_y$). The application has a *time sensor* that updates the $time(HHMM)$ entry continuously, and a *location sensor* that updates the $place(X,Y)$ information as the user moves.

Finally, the belief base contains a catalogue of user's meetings represented as entries in the form:

$$meeting(m_1, time(t_1), place(X_1, X_1)), c_1^1, \ldots, c_1^n\}.\\ meeting(m_2, time(t_2), place(X_2, X_2)), c_2^1, \ldots, c_2^n\}.$$

$\ldots$
$meeting(m_n, time(t_n), place(X_n, X_n)), c_n^1, \ldots, c_n^n\}.$

Like in a realistic environment, there is a notification process to inform the user about upcoming meetings. If he decides to attend that meeting, then he should be moving towards the designated venue at the right time. The application can assert this situation if the condition for the meeting is "in" the window of opportunity, with regards to time and space. The goal of the application is to start to download the files for that meeting proactively.

Every time either $time(t)$ or $place(X, Y)$ gets updated, the *context observer* will calculate the proximity conditions for desires – because these entries are part of the conditions and, thus, have relevance filters associated to them.

Hence, the *function to calculate proximity* of the condition C' of the desire $\langle C', \varphi', redo' \rangle$ to work as follows:

- find the entries $time(t')$[7] in the list of terms in C, if any; if there is more than one term $time(t')$, then the algorithm considers that the proximity cannot be calculated; in this case, the decision is taken based on whether C holds with the belief base;
- run the same process for the entries $place(x', y')$, if any present; again, it is expected that only one entry is present, otherwise the proximity cannot be calculated;
- then, for the remaining conditions $c'_1, \ldots, c'_n$ in C, check whether they hold with the belief base;
- if true, then the proximity condition can be calculated as follows:

  current position: $place(x, y)$
  destination position: $place(x', x')$
  physical distance: $d_p =\mid x - x' \mid + \mid y - y' \mid$

  current time: $time(t)$
  destination time: $time(t')$
  time distance: $d_t =\mid t - t' \mid$

  window of opportunity: $woo(\varphi', d'_p, d'_t)$
  calculate: if $d_p <= d'_p$ & $d_t <= d'_t$, then return "in", else return "out".

[7] *Time operation*: this operation can be implemented by a programming construct that scans all terms in the list of PROLOG terms that compose the object *Condition* (see Table 5.1).

Of course, this approach is a *reduction* tailored for the specific purpose of this class of application and scenario. It works as a reference for extended implements. The proposal has several shortcomings, though. For example:

- the terms $time(t)$ or $place(X, Y)$ must be considered as *reserved words* while coding the conditions;
- the representation is restricted and not flexible, with the terms representing very specific information related to time-space conditions, although it should be possible to extend these operators in order to accommodate more meaningful representations, and;
- finally, it requires that only one term is present to avoid inconsistencies. If more than one term is present then the method cannot consider the time-space condition. This situation is a side-effect of the limited scope of the two operators and a more flexible representation should help to resolve the situation.

Although simplistic, I argue that this solution is sufficient for simple demonstrations and restricted problem scenarios. Of course, it is not ideal for a real world application, as it requires the programmer to observe the above restrictions, which imposes a cost at programming time.

Example 10. (proximity calculation)

For example, let us consider that the application has two meetings scheduled like:

$meeting(m_1, time(1000), place(3, 3)), c_1^1, \ldots, c_1^n\}.$
$meeting(m_2, time(1200), place(8, 8)), c_2^1, \ldots, c_2^n\}.$

Let us say that the current position is $place(5, 5)$ *and* $time(0950)$. *In addition, the default values for the window of opportunity is* 3 *units of space and* 5 *units of time, represented by the entry:* $woo(M, 3, 5)$. *The situation is evaluated as "out" of the window of opportunity (WoO) for both meetings.*

Assuming that the user is stationary (e.g. he is working at his office) as the time progresses, when it is $time(0955)$ *the time condition is within the WoO for* m_1, *however the space condition, if calculated with the current parameters, results in* 4 *units of time, i.e. "out" of the space condition to deliver the information. That is:*

current position: $place(5, 5)$
destination position: $place(3, 3)$

physical distance*:* $d_p =\mid 5-3 \mid + \mid 5-3 \mid$ *= 4*

current time: $time(0955)$
destination time: $time(1000)$
time distance*:* $d_t =\mid 1000-0955 \mid$ *= 5*

window of opportunity: $woo(m_1, 3, 5)$
calculate: as $(d_p = 4) > (d'_p3 = 3)$*, then return "out".*

That is, if the user remains stationary at $place(3,3)$ *and does not move to the meeting venue, then the application does not download the files, even when the meeting time approaches. That is the expected behaviour of the application, because downloading the files for a meeting for which the user has been notified but does not seem intent on attending would be a waste of resource.*

However, let us say that at $time(0958)$ *the user decides to move towards the meeting venue. At* $pos(5,4)$*, the condition is evaluated as follows:*

current position: $place(5,4)$
destination position: $place(3,3)$
physical distance*:* $d_p =\mid 5-4 \mid + \mid 5-3 \mid$ *= 3*

current time: $time(0958)$
destination time: $time(1000)$
time distance*:* $d_t =\mid 1000-0958 \mid$ *= 2*

window of opportunity: $woo(m_1, 3, 5)$
calculate: as $(d_p = 3) == (d'_p = 3)$ *and* $(d_t = 2) < (d'_t = 5)$*, then return "in".*

The *context observer* signals the planning module to start the execution of $downloadFiles(m_1)$. The application has 3 space units to download the files and deliver the information before the user reaches the destination.

For a realistic application, the running environment provides the interfaces to collect and calculate time and location information.

5.4.2 Calculating Plan Costs

I mentioned before that the process "*[has] a method* $cost(pg_{body})$ *that calculates the cost of executing this PG-Rule in the current environment; this method is implemented as a practical solution*".

I recall that the cost of a plan π is the sum of the costs of the basic actions, plus the cost of the abstract plan π_s in the tail of the plan's body. However, in the BDI-model, the planning process behaves according to the *think-act* methodology. That is, it is not forming the whole plan ahead of execution. Hence, it is not trivial to apply this recursive formula to plan cost calculation, because the element $cost(\pi_s, C, n)$ cannot be immediately quantified.

Therefore, I provide a practical solution for the function $cost(PGs)$. It receives the set of selected plan rules from the step "*Find Possible Plan Rules for* T_h" (see Figure 5.6, (l) Plan Execution) and applies the following algorithm to look for the entry with "'*lower cost*" .

- start $LowerCost = \infty$ and $CheaperPlan = \emptyset$;
- for each entry "$\varphi' \leftarrow C_i \mid a_i^1, \ldots, a_i^n, \pi_i$" $\in PGs$, where $a_i^j \in BAction$ and $\pi_i \in \mathsf{Plan}$ do;
- for each action $a' \in \{a_i^1, \ldots, a_i^n\}$ do;
- find $cost(a', n')$; this lookup is implemented as a PROLOG query to the belief base; if no entry is found, considers $n' = 0$ (no cost associated to the action);
- add n' to the total cost of the plan rule: $tCost_i+ = n'$; close the inner for-loop, and;
- if $tCost_i < LowerCost$, then $LowerCost \leftarrow tCost_i$ and $CheaperPlan \leftarrow$ "$\varphi_i \leftarrow C_i \mid a_i^1, \ldots, a_i^n, \pi_i''$
- close the outer for-loop;
- return $CheaperPlan$.

Essentially, the process compares the costs of the basic actions that lead the plan body and selects the plan with the lowest sum of the costs of individual basic actions.

That approach is controversial, however, as it does not seem equitable to compare plan rules that have different number of basic actions. That is, for example, if the set PGs below have three plan rules like:

$pr_1 = \varphi' \leftarrow C_1 \mid a_1^1, \ldots, a_1^2, \pi_1$"
$pr_2 = \varphi' \leftarrow C_2 \mid a_2^1, \ldots, a_2^5, \pi_2$"
$pr_3 = \varphi' \leftarrow C_3 \mid a_3^1, \ldots, a_3^{10}, \pi_3$"

The plan rule pr_1 has two actions in front of the sub-goal, the plan rule pr_2 has five actions, and pr_3 has ten actions. Of course, it is more likely that the first plan rule has the lower cost because it has fewer actions to sum up.

One might argue that it makes more sense to sum up the costs for the same number of leading actions in each plan rule. That is, in the example above, only the first two actions in each plan rule would count towards the cost calculation. In addition, one could claim that this approach leaves an open variable whereas it is not possible to calculate the costs of the tailing sub-goals.

This solution is a conscious option for a simple way to calculate the cost based on all the information that is available – i.e. the complete set of actions in each plan rule. In any case, there is no way to assert the impact to resolve the tailing sub-goal in the cost calculation. Moreover, it is possible that the two actions in pr_1 are very expensive and the ten actions in pr_3 are cheaper (than the first two actions), so even though the later has more actions it is still a cheaper sub-plan than the former.

Therefore, I acknowledge that this method implements a *guessing* based on the assumed parameters. Such as the alternative above, there are no guarantees it will result in the lowest cost in the overall plan resolution. Most likely, it will lead to minimal locals, however, I argue that this approach suffices for the simple demonstrations intended by this work.

5.4.3 Executing External Actions

I recall that 2APL provides the representation of an external action $\langle externalaction \rangle$ as an expression of the form:

$$@Env(ActionName, Return)$$

Where *Env* is the name of the agent's environment, implemented as a Java class in current 2APL implementation; $ActionName$ is a method call (of the Java class) that specifies the effect of the external action in the environment; *Return* is a list of values returned by the corresponding method.

The execution of these actions changes the external environment. In 2APL, the application executes the method $ActionName$ in the Java class *Env*. However, this solution is not portable into the prototype as the running environment (i.e. *Java mobile run-time environment*) does not support the so-called *reflection*: that is, the feature of Java language that allows a program to refer to a method based on its name in a string format.

I introduced a practical solution based on the presence of *actuators* (see Figure 5.2) that are indexed by *labels*, in string format. These structures are implementations of an *interface* class that provides the common programming interface for method execution. The programmer must implement the functionality and configure the actuators during the initialisation process.

The execution of $@Env(ActionName, Return)$ leads to the following actions:

- find the *actuator* indexed by the name *Env*;
- execute the method $execute(ActionName)$ in this Java class;
- unify the returning result from that method execution to *Return*.

The action to execute the method $execute(ActionName)$ is wrapped around a $try - catch$ statement to capture execution events. Thus, in case either the actuator does not exist or the method throws an event, the process generates an *exception* that is captured by the planning module and interpreted as a *physical exception*. This exception will be forwarded to the *learning module* and cascade down to the *function for plan revision* in the workflow.

5.4.4 Technical Solutions

There are several technical solutions adopted for the implementation of the prototype. I highlight that implementation details are out of the scope of this presentation. The application code is released as an open-source, which can be analysed and extended. I refer to project's web-site for more information.

Running Environment and Field Technologies

The prototype is developed upon the *Java 2 Platform Micro Edition* (J2ME) platform[8]. This is the version of Java designed for consumer and embedded devices, such as PDAs and mobile phones. J2ME uses the same language and tools as the desktop version of Java, which is J2SE. However, it provides a reduced application programming interfaces (APIs), optimised for the memory, processing power, and I/O capabilities of specific devices, as described in Riggs et al. [2001].

The strength of J2ME is industry adoption, its *wide availability* in the market, and *compatibility* with device standards. The option to use J2ME ensures that the application is deployed onto a myriad of existing devices.

Moreover, J2ME provides a network of specifications and technologies to access mobile device resources. These programming interfaces and libraries

[8] *J2ME web-site*: accessible at http://java.sun.com/j2me.

are described by the *Java Community Process Program*[9], and provide support to commonly available physical interfaces, such as device displays, keyboards, sound, location-based systems, and communication.

PROLOG Engine

As mentioned the implementation works upon a PROLOG Engine that provides the logical basis. This structure provides the basic components for the elements of the solution (see table 5.1) and the operational support for several functionalities, such as the selection of plan rules, and support to queries to the belief base (which is a PROLOG Knowledge Base).

I provide a reduced PROLOG engine optimised for *Java mobile run-time environment*: the PROLOG-M. This engine was developed based on the W-Prolog project[10].

However, that product was heavily re-engineered for the purpose of this work. The implementation provides several methods to facilitate the integration of 2APL programming structures. For example, the query methods provide the structure to return the set of *substitutions* θ, which are required to implement the transitions of the operational semantics (see Section 3.4)[11].

Moreover, the implementation provides the programming constructs to access elements of the unification stacks. This is a requirement for the process of resuming processing threads, as elements of the unification stack must be adjusted in some cases.

5.5 Conclusion

This implementation demonstrates that the specifications are detailed enough to be reproduced as a software application. The development followed the guidelines proposed by the design model in the previous chapter, and answers the following software engineering-related questions:

- (i) how to represent and store agent's status information?

  > The architecture introduces the specifications of knowledge bases and structures for the representations.

[9] *JCP web-site*: accessible at http://jcp.org/.

[10] *W-Prolog*: web-site at http://goanna.cs.rmit.edu.au/ winikoff/wp/, accessed in May-2008.

[11] *Substitutions*: for example, in the transition in the plan selection function in Definition 29, it is required that the sets of substitutions η_i (from goal resolution) and θ_i (from condition resolution) are accessible and passed through several steps.

- (ii) what are the expected operations by each module?

  The architecture proposes to segment the operations into the modules described in the *design model*. The operations were detailed in the previous chapter and can be re-used in this implementation.

- (iii) how to implement the interfaces between the modules?

  The workflow represents the event-passing structures between the modules.

- (iv) how to implement the interfaces to external elements?

  The architecture proposes interfaces to field technologies for sensors, actuators, and parser; practical solutions for this integration are presented.

There are several challenges associated with the implementation. Firstly, the design was integrated with operation semantics and programming constructs of the existing language. This is a requirement for re-usability of existing specifications and field adoption.

In addition, the implementation provided solutions for practical problems relating to calculating proximity (for the context observer module); calculating plan costs; pausing and resuming processing threads (both for plan execution module), and enforcing the operation constraints. The prototype provides practical solutions in form of programming constructs, thus preserving the 2APL's operation semantics.

Issues relating to the physical implementation in mobile devices have been mitigated by recent technological developments. There are still some restrictions inherent with mobile computing related to small display sizes, the requirement for processing optimisation (to reduce battery utilisation), interfacing with external technologies, and limitations in programming libraries, however, I argue that they do not impose a major challenge, as the current high-end mobile computing devices provide the computing resources required by the target applications.

5.5.1 Support by 2APL and Extensions

The prototype integrates an existing agent programming language, the 2APL. This language provides the agent-model elements – i.e. knowledge bases and operational semantic – and the definitions of the deliberation cycle.

The use of 2APL programming language provides a number of desired features, such as the explicit definition of the common elements, and support of event handling, exception handling, and plan revision. In addition, the formal semantics provided by the 2APL language definitions facilitate the process of defining the cycle's operations. These elements are highlighted in Figure 5.6 and summarised below:

- the definitions and structures of the knowledge bases, i.e. the belief base, desire base (goal base in 2APL), plan rules base, and capability base;
- the operational semantics for plan formation and execution;
- the operational semantics for processing external and internal events, and;
- the operational semantics for the plan revision process.

These structures correspond to the core processing functionality, which coordinates the applications' status representation and decision taking process.

On the other hand, the proposed extensions work as a shell around the agent-model core. These structures exist to (i) enhance the reactiveness to external events (i.e. the sensor module), and (ii) adjust the operational parameters to enhance performance (i.e. the learning process). In this work, I purposefully focused on the first structures, centring discussing around the question on *when* to reconsider intention deliberation. However, issues of learning are out of the scope, although I acknowledge their importance towards better performance and more proactive systems.

In the final design the extensions are responsible for the following:

- sensing external events;
- filtering relevant external events, which is events that can potentially affect any of the current processing threads;
- calculating the proximity of the new situation to given opportunity criteria, that is evaluating if the current situation is in or out an *window of opportunity*;
- signalling the deliberation cycle – part of the agent-model core – to start, pause, resume an intention deliberation, and;
- the practical solutions that enable starting, pausing and resuming planning threads, such as initial intention formation, planning thread control, unification stack control, and enactment of the plan revision process.

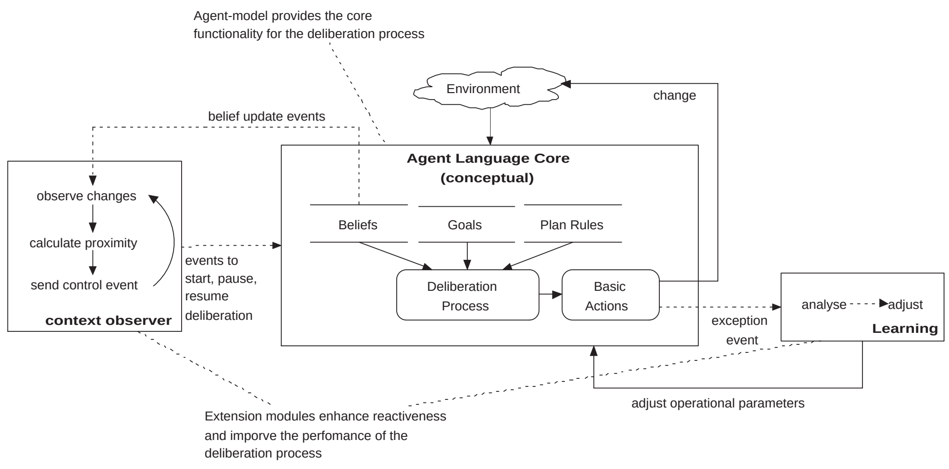

Agent-model provides the core functionality for the deliberation process
Environment
change
belief update events
Agent Language Core (conceptual)
observe changes
calculate proximity
send control event
context observer
events to start, pause, resume deliberation
Beliefs
Goals
Plan Rules
Deliberation Process
Basic Actions
analyse
adjust
Learning
exception event
adjust operational parameters
Extension modules enhance reactiveness and imporve the perfomance of the deliberation process

Figure 5.7: Integrating to Other Agent Programming Languages

The integration with 2APL agent programming language was feasible due to its support for common programming constructs, and the support provided by the running environment. I argue that the extended functionality can be integrated into different agent-model languages, however, and elaborate on this idea in the next sub-sections.

5.5.2 Extensibility to Other Agent Programming Languages

In Section 5.2, I explained why I chose 2APL to provide the core functionality for my prototype. I argue, however, that the model could be integrated into a number of existing BDI-model agent languages, which I labelled as "candidate languages", as long as they offer the following requirements:

- the common BDI-model elements, such as the representation for beliefs, desires (or goals), intentions, plans, and basic actions;
- the operation semantics for the internal operations, and;
- the representation of the processing elements that allow them to be controlled, i.e. started, paused, and resumed.

That is, the agent-model provides the core functionality for the deliberation cycle; which is the functionality being re-used by the prototype's implementation.

However, considering that the content observer and learning functionalities are implemented as separated modules, as depicted in Figure 5.7, these modules can be integrated to different agent-model cores. The question is *how* to implement this integration? The *requirements* for the integration are:

- it must provide the representation for intention deliberation process control that accommodates the running and paused states for processing threads, or;
- in case an existing implementation is to be re-used, it must provide the programming interfaces to control individual processing threads;
- it must provide the programming constructs to capture exceptions generated by basic action execution in order to initiate the reactive learning process;
- it must provide the interfaces to adjust the operational parameters (e.g. basic action costs, and goal priorities), and the agent knowledge structures (e.g. beliefs, plan rules, and desires) in order to operationalise the learning functionality, and;
- it must provide or allow to incorporate the functionality for processing thread validation in order to resume the processing thread, that is to verify whether a thread is still valid in the current situation and adjust its operation parameters (e.g. unification stack), when required.

The integration offers numerous advantages: it enhances the reactiveness to environmental changes in a controlled, rationalised way, that is, the context observer provides the solution to determine *when* to reconsider the processing without having to incorporate special structures on the plans' conditions, and rationalises the reconsideration process by implementing filters for the relevant (i.e. potentially impacting) environmental changes.

I argue that this structure adds value to the existing platforms by providing a solution to the problem of *when* to revise intention deliberations. I consider extending implementation to other agent languages to be outside the scope of this work, however, this could be a future work.

Next, I elaborate on how these structures can also be used for other knowledge domains beyond mobile services.

5.5.3 Extensibility to Other Application Domains

In this work, I am proposing to implement *Mobile Personal Assistants* (MPAs) as a solution for location-based information systems. Throughout this work, I have been exploring *what* must be extended in the current agent-models to support the development of these applications.

I argue that the proposed model could also be applied in different knowledge domains to implement applications that operate in high-dynamic environment, make use of context information for decision taking, and implement both reactive and proactive behaviour. The main difference is that the context observer would need to be equipped to realise and make sense of different dimensions of context information.

What are the different dimensions of context information? Hong and Landay [2001] has described context as knowing the answers to the "W" questions, such as "Who is speaking". Graham and Kjeldskov [2003] proposed segmenting context information into eight dimensions, with "W" questions associated with them as:

1. Time: When?
2. Absolute location: What position?
3. Relative location: Where?
4. Objects present: What else?
5. Activity: What work?
6. Social setting: Who?
7. Environment: What condition?
8. Culture: What culture?

That work describes the dimensions as:

> "*Thus the first dimension addresses the time of day, the second the origo's position, the third the origo's position in relation to other people or objects, the fourth whether other devices are in the same space. The fifth dimension captures the goals, actions and operations of the origo the sixth the number of people present and the social occasion. The seventh dimension considers the physical environment and the eighth the cultural environment.*"

Therefore, the context observer must be equipped to calculate proximity beyond time-space conditions, which are, in fact, a sub-set of the context

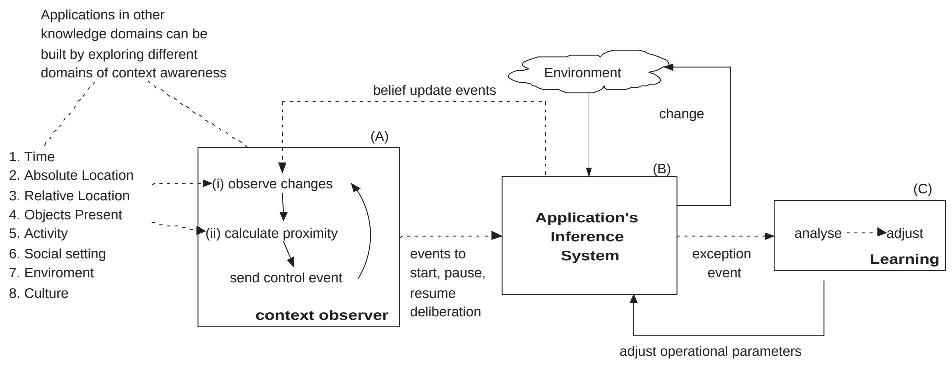


Figure 5.8: Extending to Other Application Domains

dimensions. It is possible to think of the *in* and *out* conditions (in relation of the window of opportunity) in terms of yes-no questions. For example: Are the objects present? Is the user engaged in the some activities? Is the user part of the social setting segments? and Are the conditions valid?

Some of these conditions relate to profiling techniques that are also applied to "profile and personalisation" solutions, such as target advertisement systems. These solutions apply data mining techniques to infer possible contexts that the user has been involved in to determine the *window of opportunity* to deliver relevant types of advertisement to the user. Therefore, it is realistic to think about the integration of the proposed technology to applications that follow the user's changing context to infer the current settings and deliver targeted information.

What are the steps to create these applications? Figure 5.8 depicts a high-level architecture for these applications. Basically, the (A) context observer is integrated to the (B) application's inference system, which is integrated to the (C) learning module. The inference system is application dependent and could be implemented, for example, by using reactive loops, such as those based on object-oriented programming. This module must implement the representation of the processing elements that allow them to be controlled, i.e. started, paused, and resumed, ideally individually.

The context observer definitions can be re-used, but they have to be adapted to explore information from the other dimensions of context awareness. I suggest that the developer needs to implement sensors that are able to:

- (i) observe changes in the other dimensions. In general, this information is already represented in the application somehow, for example, through the presence of some object, the social setting, or the culture in which the user is inserted. This function would have to be implemented in a way that it perceives variations in these conditions and passes this information to the function that calculates proximity, and;

- (ii) calculate proximity, such that it can infer the in-out condition in relation to the window of opportunity for the other dimensions of context information. As mentioned, it is possible to think of these conditions in terms of yes-no questions, as for example:

  1. Time: Is the task still feasible in time?
  2. Absolute location: Is the position still valid?
  3. Relative location: Is the position still valid?
  4. Objects present: Are the objects $o_1, \ldots, o_n$ present?
  5. Activity: Is the user engaged in the activities $a_1, \ldots, a_n$?
  6. Social setting: Is the user part of the social setting segments $ss_1, \ldots, ss_n$?
  7. Environment: Are the conditions $c_1, \ldots, c_n$ valid?
  8. Culture: Is the user immersed in the cultural setting $cs_1, \ldots, cs_n$?

The *proximity* could be derived from the combination of the conditions inferred from the questions above. Then, the context observer would signal the inference system to start, pause, or resume processing threads. The inference system must provide the same *requirements* for the integration, as described in the previous sub-sections, that is:

- it must provide the representation for intention deliberation process controls that accommodate the running and paused states for processing threads, or;
- it must provide the programming constructs to capture exceptions generated by basic action execution in order to integrate the reactive learning process;
- it must provide the interfaces to adjust the operational parameters (e.g. basic action costs, and goal priorities), and the agent knowledge structures (e.g. beliefs, plan rules, and desires) in order to operationalise the learning functionality, and;
- it must provide or allow incorporation of the functionality for processing thread validation in order to resume the execution.

Such as when integrating to other agent-languages, presented in the previous sub-section, these structures add value to the existing applications by enhancing the reactiveness of the application to relevant (i.e. impacting) changes of the environment, and provides a simple solution to the problem of *when* to reconsider the current processing. In the existing solutions, these elements are usually embedded in the processing code, in form of if-condition-then-revise statements. I argue that having this structure separated from the main inference loop provides the following benefits:

- it simplifies the coding process;
- improves re-usability, and;
- enhances overall application performance, boosting reactiveness to external events, while filtering only the relevant changes for processing consideration.

I present a set of case study scenarios built on top of the prototype application in the next chapter.

Chapter 6

Case Studies

In this chapter, I present a set of qualitative and quantitative analyses through case study scenarios.

6.1 Introduction

In the previous chapters, I introduced an extended deliberation cycle that takes advantage of environmental events to promote reconsideration of the deliberation process. Applications developed upon this model provide better performance when operating in highly dynamic environments, such as mobile services.

However, up to this point, the assertion on performance enhancement has been concluded based on intuitive analysis of isolated elements. To prove these arguments with comparative analysis, I propose to run a series of experiments. I have prepared a number of case studies where I analyse the performance of the application when equipped with the proposed model *versus* variations of degrees of reactiveness, as proposed in Kinny and Georgeff [1991] – i.e. *bold agent* where the context observer is disabled, and *cautious agent* where the filters of relevance are disabled. These exercises endeavour to:

- measure the effectiveness of the proposed model in a simulation environment;
- investigate how features of the proposed model contribute to effective agent behaviour, and;
- compare the properties of different strategies for reacting to environment changes.

The experimental system is based on the prototype implementation presented in the previous chapter operating on an extension of the TileWorld Pollack and Ringuette [1990] with the elements of mobile service. I call this testing environment "the MobileWorld".

The chapter is structured as follows. In the next section, I review the essential features of the experimental system and introduce the terminology used in the analysis. Section 6.3 presents and analyses the results of the different experiments, before the chapter concludes with a discussion of the results in section 6.4.

6.2 Experimental System

I propose to run a number of experiments to prove the performance advantages of the model. I will run variations of the deliberative behaviour in the same controllable environment to reproduce similar situations - and compare the results. This way I can infer that any performance variations are due to deliberation behaviour, and not environmental configurations.

I analyse the requirements for the test environment below and introduce the solution in the next sub-section.

6.2.1 Requirements

First, I suggest the need for a test environment that is able to reproduce realistic operational conditions, and the fundamental issues present in mobile services. Moreover, it must provide the facilities to assert the benefit of the proposed model. The environment must support tools that measure computation efficiency and performance in order compare the results for multiple test cases with different application configurations. Considering these guidelines, I introduce the following requirements for the test environment:

- *reproduce constantly changing context.* This recreates one of the main elements of dynamic environments; in mobile service, the environment gets update by both users' activities and other elements of the environment. Thus, the parameters must be controllable to measure the influence of different environment configurations.

- *reproduce spontaneous interactions.* This recreates the condition where unexpected events can occur; moreover, it must consistently reproduce these situations in order to support comparative analysis in similar conditions.

- *reproduce instabilities*, reproducing common issues in mobile services such as transient tasks, and connectivity fluctuations. Again, the test environment must consistently reproduce these situations in order to support comparative analysis in similar conditions.

- *test different agent configurations*, which supports asserting the benefits of the proposed model by comparing the results of different agent configurations. The test environment requires the ability to enact distinct reconsideration strategies, i.e. bold agents, cautious agents, as described in Kinny and Georgeff [1991], and the implementation based on the model agent.

- *separation between user's and mobile assistant's behaviour*, where the goal of the experiments is to measure the performance of MPA application in reaction to environmental changes, which includes unexpected events driven by the user behaviour, such as mobility, and the user engaging to different activities.

- *provide facilities for ensuring application performance*, so I can compare the difference in computational efficiency and performance between different scenario and agent configurations.

The requirements to reproduce constantly changing context, spontaneous interactions, and instabilities are essential as the simulation environment must replicate the characteristic of "real world" mobile services. There must be a mechanism to control the degree of dynamism of the environment. That is, it should be possible to control the frequency of these events. Hence, one can assert the influence of different environment configurations on the application's performance.

In addition, the requirement to separate user's and mobile assistant's behaviour is important for avoiding confusion between variations of user behaviour and its impact over the application's actions. Experiments like the one presented in Schut et al. [2004] analyse the performance of the different reconsideration strategies from the view of one agent action in a simulation model. That work considers two different models of intention reconsideration, based on discrete deliberation scheduling, which analysis in terms of conventional decision theoretic models of optimal action, and; partially observable *Markov decision process* (POMDP), whose solution implies in finding an optimal intention reconsideration policy.

I acknowledge that this model can be extended to encompass different features of complex mobile services; however an alternative is to reduce the configuration's scope. These approaches lead to unrealistic, non-intuitive testing environments, whose results can not directly relate to real world situations. Therefore, the simulation environment must support separate modelling of the user behaviour and mobile assistant application, while considering the interactions between these components.

In the next sub-section, I introduce the simulation environment and describe how it fulfils these requirements.

6.2.2 The Environment

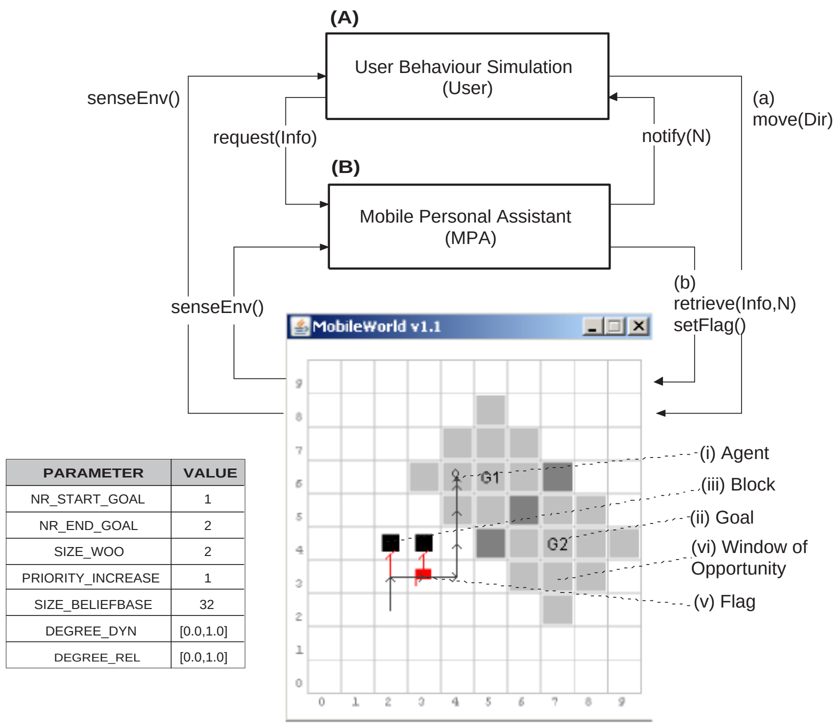
To conduct the experiments, I need a controllable environment that allows me to evaluate the effect of different deliberation strategies. I argue that "real world" case studies are not an option due to the difficulty of implementation, which is both time and resource consuming. In addition, I must be able to reproduce the same tests repeatedly, with small variations of configurations – e.g. variations of the degree of dynamism, which is not always feasible in the real world.

Therefore, I introduce a simulation environment that captures the essential features of real-world domains while permitting flexible, accurate, and reproducible control of the world's characteristics. As stated in the introduction of this chapter, the experimental system is based upon a MobileWorld extension of the proof-of-concept implementation of the proposed model.

In essence, the MobileWorld is a 2-dimension grid where the elements of mobile service environments are represented, as depicted in Figure 6.1 and described below:

- (i) *agents* represent the mobile users and mobile personal assistant;
- (ii) *goals* represent the positions that mobile users desire to achieve;
- (iii) *blocks* represent the tiles that the agent cannot move to, which the agent can sense the blocks before it tries to move into that tile;
- (vi) *window of opportunity* denotes the "opportunity area" to deliver information for a goal, and;
- (v) *flags* mark the position where the mobile personal assistants deliver the information for a goal.

Goals appear in randomly selected squares with a configured priority and assigned window of opportunity. Every time the agent reaches a goal, it is removed from the board. In addition, the board provides variables to control


(A)
User Behaviour Simulation (User)
senseEnv()
request(Info)
notify(N)
(a) move(Dir)
(B)
Mobile Personal Assistant (MPA)
senseEnv()
(b) retrieve(Info,N) setFlag()
MobileWorld v1.1
G1
G2
(i) Agent
(iii) Block
(ii) Goal
(vi) Window of Opportunity
(v) Flag

PARAMETER	VALUE
NR_START_GOAL	1
NR_END_GOAL	2
SIZE_WOO	2
PRIORITY_INCREASE	1
SIZE_BELIEFBASE	32
DEGREE_DYN	[0.0,1.0]
DEGREE_REL	[0.0,1.0]

Figure 6.1: The Environment

the maximum number of goals on the board. In addition, it must have a number of "bogus" environmental facts that are not related to the conditions of the running elements; thus, it is possible to configure different degrees of relevance in belief base updates based on updates of these entries. The list of environmental *parameters* is summarised below in Table 6.1.

The environment works as follows:

1. the controlling logic selects how many elements of the environment will change based on the configurations;

2. it distributes these changes between the bogus and relevant changes based on $DEGREE\_REL$;

3. it updates the representation of the environment, and;

4. the agents perceive the modifications at each time cycle via the sensor module.

I argue that the configurations for goal dynamics are enough to demonstrate different scenarios with variable degrees of dynamism. For example, to reproduce a static world one can set $NR\_START\_GOALS = NR\_END\_GOALS$.

Parameter	Description
NR_START_GOALS	Determines how many goals are placed on the board when the run starts
NR_END_GOALS	Determines how many goals are placed to the board in total
SIZE_WOO	Determines the size of the windows of opportunity
PRIORITY_INCREASE	Determines the increment for the goal priority for new goals asserted to the board; that is, every time a new goal is asserted it is assigned with a higher priority of this factor
SIZE_BELIEFBASE	Determines the size of the belief base, which is completed with bogus entries; this parameters is used in conjunction with the degree of relevance, introduced below, to measure the computation performance for different belief base configurations
DEGREEDYN	Determines the rate of world change, which is related to adding or moving a goal, or changing bogus entries, depending on the configuration below; 0.0 means that the environment never changes, 0.5 means that there is 50% chance that environmental variables change; 1.0 means that at least one element of the environment changes at each timer cycle
DEGREE_REL	Determine the degree of relevance of world change, i.e. what is the percentage of changes that affect the conditions of the existing elements; 0.0 means the only non-relevant entries change; 0.5 means that 50% of changes are related to non-relevant changes; 1.0 means that only relevant entries change each timer cycle

Table 6.1: Parameters of the Simulation Environment

On the other hand, highly dynamic environments can be created by setting $NR\_START\_GOALS \neq NR\_END\_GOALS$ and $DEGREE\_DYN = 1.0$. I shall exploit the use of these configurations in the case studies.

Finally, some *assumptions* on the board's behaviour are described below.

- the agent can move only one square per timer cycle;
- the agent can retrieve only one piece of information per timer cycle;
- the board can add only one new goal per timer cycle;
- every time a new goal is added to the board, a message is sent to the existing agents describing its name, position, window of opportunity size, and priority.

The first three assumptions create the baseline for the relationship between dynamism and time. The last assumption enforces consistency between the agents' knowledge of the world and the external environment. These assumptions exist mainly to simplify the development process.

6.2.3 The Agents

Figure 6.1 depicts the various programming interfaces that the agent can use to control the environment. They include two sets of actions: (a) the actions to change the agent's position, and (b) the actions to retrieve information. Thus, there are two classes of agents associated with each class of actions. This configuration supports the requirement for separation between user's and mobile assistant's behaviour. The agents are depicted in the diagram and explained below:

- (A) *user behaviour simulation* (UBS) controls the actions of the "user" on the board; for example, moving toward a goal or changing tasks;
- (B) *mobile personal assistant* (MPA) controls the actions for information retrieval, processing, and provisioning. Agents are interconnected through the communication structure: the UBS sends requests for information to the MPA, which retrieves this information from the board and communicates it back to the UBS.

The UBS is an agent program with the desire to reach the existing goals following the order of priority. The plan rules describe how to move the agent towards the goals. The context observer is fully enabled reflecting the human capability of perceiving and reacting to environmental changes; although I highlight that modelling human behaviour is out of the scope of this work. The implementation provides a simple, reactive strategy to detect and move towards goals, which suffices for the purpose of the case studies.

The UBS agent is aware of the goals it is trying to achieve and the windows of opportunity. If it receives a notification for a goal while out of the window of opportunity, then it increments a *disappointment gauge*. Thus, it is possible to measure the level of effectiveness of the MPA agent as reversely proportional to the disappointment level caused for the UBS agent. This reflects the real world situation where the UBS agent perceives notifications delivered out of the window of opportunity as inconvenient and coming from a “poorly developed application”.

The MPA agent simulates the behaviour of the application that implements the advisor role to the UBS agent. It consists of the desires and plan rules for task and time management. It is motivated by the *delegative* model of interaction between the user and the application, where the user decides what needs to be done and what tasks he feels comfortable allocating to the system. The MPA operates in a fairly autonomous manner with regards to the UBS agent’s behaviour, but interacts to solicit necessary information and to inform and confirm decision. The MPA is supposed to adapt its behaviour in accordance with the user behaviour and deliver the right information at the right time. The interaction between the MPA agent and the UBS agent is implemented through the communication structure *via* message passing.

For the test cases, I am focusing not on how to program the MPAs but rather on how the same program reacts in different agent configurations. Hence, I provide a simple agent program that contains the following elements:

- the information about upcoming meetings, providers, etc. This information is initialised in the agent’s belief base and updated *via* communication from the board about new goals, reproducing the situation where multiple agents are exchanging knowledge and information.

- the desire to enact the rules for task and time management, observing the conditions imposed by the board configuration, i.e. goals and windows of opportunity.

- the plan rules to enact the role of task management, such as to organise, filter, prioritise, and oversee tasks’ execution on behalf of the user. In the current model, the task are associated with information retrieval and event notification, however, these rules can be extended to more elaborate problem-solving entities, such as planning, scheduling, communication, and coordination.

- the plan rules to enact the role of time management, such as proposed by Berry et al. [2006] "*to personalised time-management support, responsive to user's needs, preferences, and adaptive to changing circumstances.*

During the MPA initialisation, the agent application loads the desires and plan rules configuration from an agent program file. Next it senses the environment through a call to the *senseEnv*() interface in the board implementation, which returns the updates in the surrounding environment that are represented in the belief base. The implementation executes one update each timer cycle. In addition, new tasks can be *communicated* to the agent via the communication structure from either the UBS agent or the board.

This configuration reproduces the real world scenario where the application is initialised by a configuration file, senses and represents the surrounding environment, and learns new desires and plan rules through the communication interface.

Variations in Degree of Commitment

One of the primary aims of this study is to investigate different aspects of agent commitment and its effect on performance. I recall that the board provides the parameters to vary the degree of dynamism of the environment; that is, the agents in MobileWorld are potentially faced with a new set of options after each step, either because the environment changes – as explained above, the agent senses the environment each timer cycle – or because new desires and plan rules are communicated.

I want to examine how this variation affects the efficiency of the agent behaviour, and conversely, how variations of reconsideration strategies cope with different degrees of dynamism. This configuration supports the requirement to allow testing different agent configurations.

In this study, I am applying the same agent configurations proposed in Kinny and Georgeff [1991] plus the proposed model. These configurations are detailed below.

The *bold agent* never reconsiders its options before the current plan is executed, as proposed in Pollack and Ringuette [1990]. In this configuration the agent does not consider the impact of environmental changes on the current deliberation. The architecture for this configuration is depicted in Figure 6.2(A); in short, the context observer functionality is disabled. This way, the planning module is not informed of changes in the belief base whilst executing a plan. As a result of this inflexibility, agents implementing this reconsideration strategy fail to operate effectively in dynamic environments, as demonstrated

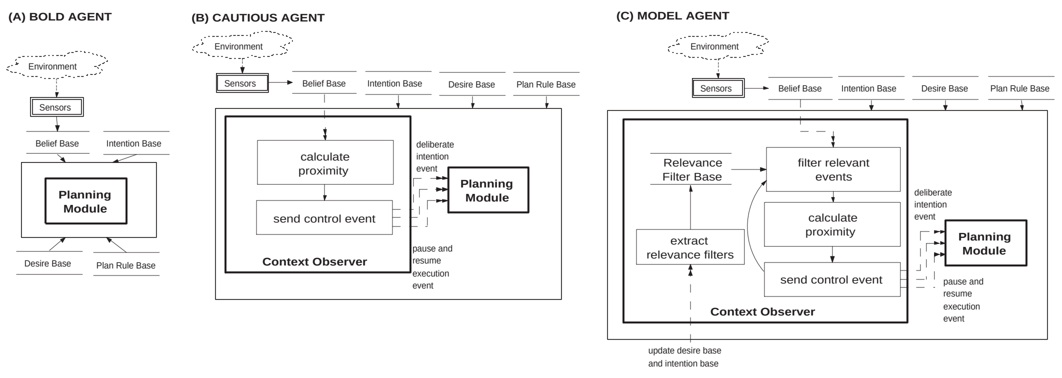


Figure 6.2: Reconsideration Strategies

in Kinny and Georgeff [1991]. In this work, I extend that study to demonstrate the consequences in effectiveness and computation performance in other scenario variations, as explained in the next sub-section.

The *cautious agent* considers the impact of environmental changes each timer cycle. In this configuration, the function to filter relevant events in the context observer is disabled. This way, the conditions for every element will have to be evaluated for any change in the belief base configuration, which implies more computational resources. This architecture is depicted in Figure 6.2(B). This is the basic configuration used by several agent implementations, such as *PRS*, *AgentSpeak/Jason* and *JACK*.

Finally, the *proposed model* configuration implements the architecture presented in Chapter 4. The detailed description of the context observer configuration was introduced in Section 4.3.1. The architecture diagram is reproduced in Figure 6.2(C).

The different configurations between cautious and bold model agents enable me to investigate how sensitivity to change could affect agent behaviour. The results are influenced by varying the parameter *degree of relevance*, which is described above. However, in this work, I am not exploiting the possibility of varying the degree of boldness by changing the observation period, as proposed in Kinny and Georgeff [1991]. My goal is to highlight the difference in effectiveness between bold and cautious model agents when operating in dynamic environments. Hence, I argue that the configurations with maximum boldness and cautiousness suffice for my experiments.

6.2.4 Performance Measurement

In the MobileWorld, time is measured based on an abstract clock. The board and the agents execute synchronously, respecting the assumptions described before regarding how much can alter per time cycle. Different degrees of dynamism and relevance for environment updates can be set by the configuration parameters described in Section 6.2.2.

The *qualitative performance* can be inferred from the convenience of the information provided by the MPA agent. I recall that the UBS agent implements a disappointment gauge that is incremented every time the MPA agent provides a notification for a goal outside its window of opportunity. Hence, it is possible to measure the level of effectiveness numerically via that counter.

On the other hand, the *quantitative performance* can be inferred by the number of operations required to resolve the game. I recall that experimental results in Pollack and Ringuette [1990] revealed that measures based on CPU-time or elapsed time were undesirable. An alternative is to measure the number of operations related to the status transitions, however, it is intuitive that some transition operations are more resource consuming than others, depending on the complexity of the evaluations taking place. Therefore, I opt for a solution based on counting the number of unifications executed by the MPA agent to resolve the environment. This is a realistic measure of computational performance, as the number of unifications can be directly mapped to the number of CPU instructions and other operations executed by the application. This element determines the impact on resource utilisation, such as battery, which is a crucial point in mobile computing. The PROLOG engine used for the prototype implementation (i.e. the PROLOG-M) provides the facility to collect the number of unifications realised during the application life-cycle.

Experimental Procedure

I highlight that during the course of a single MobileWorld game, the agent's effectiveness and performance fluctuates due to the random variations in goal distribution and belief base updates. Thus, across different but statistically similar games, I observed small variations in agent effectiveness and performance, which declines for larger games, i.e. games with more goals and belief base entries.

Measuring the average of the ε and the number of unifications shows that the results stabilise and are consistent when considering the average from 12 or more repetitions. Hence, I consider an extrapolated average of 20 repetitions for each test.

Moreover, I propose to study the impact of variations of degree of dynamism (DD) and degrees of relevance (DR) upon the effectiveness and computational performance. This is because mobile services exist in an environment where $0.25 \leq DD \leq 0.65$ and $0.25 \leq DE \leq 0.65$. The experiments in

Kinny and Georgeff [1991] operate in $DR = 1.0$, so they cover only a subset of the options being tested in this study. Finally, I argue that several experimental studies consider highly dynamic environments whose $DD \geq 0.9$ and $DR \geq 0.9$, which are not realistic for real world applications. Therefore, this study is covering a number of application domains by testing the behaviour when $0.0 \leq DD \leq 1.0$ and $0.0 \leq DR \leq 1.0$.

I consider $DD = 0.4$ and $DR = 0.4$ as the standard average values for mobile services. I argue that these values reflect real world conditions where the MPA represents a large amount of information about the world, but not all get updated periodically. In addition, not all of the updates are relevant.

Next, I introduce the case studies and analyse the results.

6.3 Results and Analysis

In this section, I present the results of several sequences of experiments aimed at investigating how the agent effectiveness and computational performance changes as a consequence of environmental condition variation. The test cases are based on the elements of the mobile service problem scenario, and I related the elements of the simulation environment with real world situations whenever possible. The parameters selected for investigation are:

- Dynamism and Relevance
- Computation Performance
- Scalability

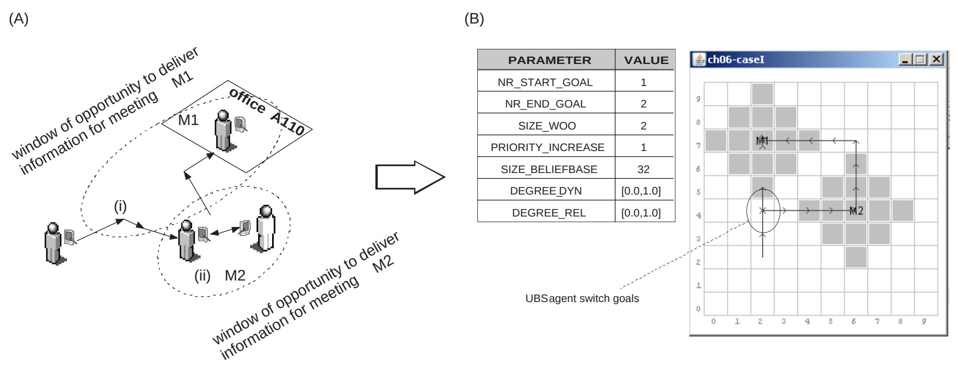
6.3.1 Dynamism and Relevance

In the first set of experiments, I focus on comparing the effectiveness when playing the scenarios with different degrees of dynamism and relevance. The experiment extends the ones proposed by Kinny and Georgeff [1991] and Schut et al. [2004] by (i) adding the proposed model's reconsideration strategy and (ii) relating the testing environment to issues of mobile services.

Predictably, the conclusions are similar to the ones reached in the above-mentioned studies; namely the ineffectiveness of bold agents in more dynamic environments. I describe the experiment below.

Testing Environment

Recall the illustrative scenario from Chapter 2, where (i) the mobile user changes tasks. The MPA agent has to switch goals and retrieve the infor-


(A)
window of opportunity to deliver information for meeting M1
Office A110
M1
(i)
(ii) M2
window of opportunity to deliver information for meeting M2
(B)
ch06-caseI
M1
M2
UBSagent switch goals

PARAMETER	VALUE
NR_START_GOAL	1
NR_END_GOAL	2
SIZE_WOO	2
PRIORITY_INCREASE	1
SIZE_BELIEFBASE	32
DEGREE_DYN	[0.0,1.0]
DEGREE_REL	[0.0,1.0]

Figure 6.3: Dynamic Environment

mation for (ii) the sideline meeting, avoiding delivering the information for the original meeting if either the user is not in its window of opportunity, or the user is no longer heading to that meeting. This scenario is depicted in Figure 6.3(A).

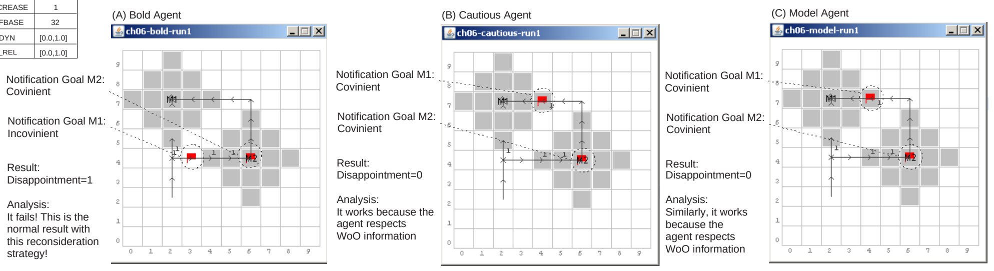
This is a typical scenario in mobile services that involves a dynamic environment, i.e. the new goal appearing on the board unexpectedly; instabilities, as the user decides to switch tasks and moves towards the sideline goal, and a requirement for adaptation as the MPA needs to adapt its execution to deliver the right information at the right time.

This scenario can be reproduced in MobileWorld as depicted in Figure 6.3(B). The MPA agent is programmed with the following elements:

- the desire to deliver the information associated with the upcoming meetings while in the window of opportunity;
- the plan rules to retrieve the information and provide the notification when the agent is in the window of opportunity;
- the belief base entries representing the environment, being updated by the sensor system, as described in Section 6.2.3.

The belief base contains the board configuration, which is being continuously updated by the sensor system, as described in Section 6.2.3. It also contains the information about the existing goals. These entries are added to the board at random times and communicated to the UBS and MPA agent via a message. Each entry contains the following information: M is the meeting identification; P is the priority; $pos(X, Y)$ represents the meeting's position; W is the size of the window of opportunity, pre-configured to 2 squares in the example; and $\{f_1, \ldots, f_n\}$ are the information elements, pre-configured to 2 elements. These entries are formatted as below:

PARAMETER	VALUE
NR_START_GOAL	1
NR_END_GOAL	2
SIZE_WOO	2
PRIORITY_INCREASE	1
SIZE_BELIEFBASE	32
DEGREE_DYN	[0.0,1.0]
DEGREE_REL	[0.0,1.0]



(A) Notification Goal M2: Covinient

Notification Goal M1: Incovinient

Result: Disappointment=1

Analysis: It fails! This is the normal result with this reconsideration strategy!

(B) Notification Goal M1: Covinient

Notification Goal M2: Covinient

Result: Disappointment=0

Analysis: It works because the agent respects WoO information

(C) Notification Goal M1: Covinient

Notification Goal M2: Covinient

Result: Disappointment=0

Analysis: Similarly, it works because the agent respects WoO information

Figure 6.4: Effect of Reconsideration Strategy

$$meeting(M, P, pos(X, Y), W, \{f_1, \ldots, f_n\})$$

At the beginning of the game the board will have the first goal, representing the original meeting. The UBS agent starts to head towards that meeting and the MPA agent will start to download the information for the meeting when the UBS agent steps in the window of opportunity. However, at some point the second goal is added with a higher priority. Then, the UBS agent will switch goals and start to head to the second goal. The question is: *how the MPA agents with different reconsideration strategies behave at this point?*

Results

Figure 6.4 presents the results from a selected goal disposition in order to demonstrate the behaviour of the different reconsideration strategies. The (A) bold agent fails to provide convenient information, delivering the notification for goal $M1$ before the notification for $M2$ and out of the window of opportunity. It happens as the agents starts to execute the plan to deliver information for $M1$ when the UBS agent enters the window of opportunity at $(2, 5)$, but it does not reconsider the processing when the UBS switches to goal $M2$ and steps out of the window of opportunity for $M1$ at $(2, 4)$. The processing for $M1$ completes at $(3, 4)$, thus delivering inconvenient information.

I highlight that the speed of generating the new goals influences the outcome of the game for the bold agents. If the goals are far apart or the second goal appears when the agent is at the border of the window of opportunity, this agent configuration always fails. It works, however, if the goals are sufficiently close or the second goal takes sufficiently long to generate. I analyse the influence of the degree of dynamism and relevance over this configuration below.

Hence, I conclude that agents implementing this reconsideration strategy fail to operate effectively in dynamic environments as a result of this inflexi-

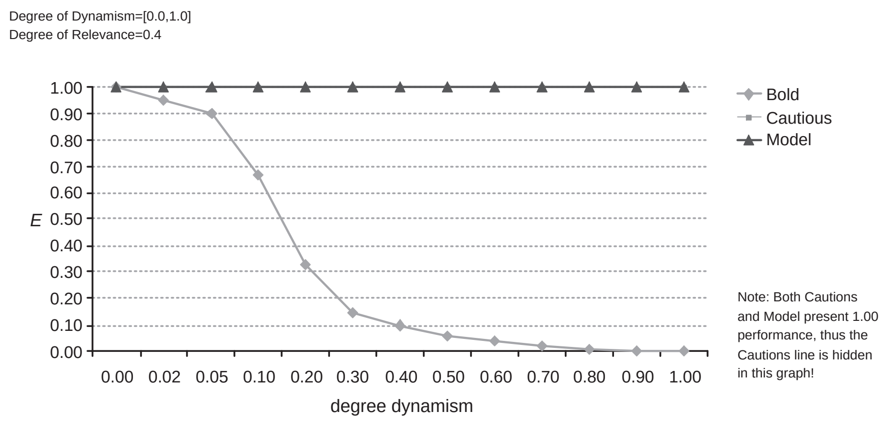


Figure 6.5: Effect of Degree of Dynamism

bility of the model. This conclusion is similar the one reached by other works such as Schut et al. [2004].

Nonetheless, both the *cautious* and *model agents* respect the window of opportunity and, in the same situation, they pause the processing for $M1$ at $(2, 4)$, start to process $M2$ when in the window of opportunity for that goal in $(4, 4)$, completing at $(4, 6)$, and resume the processing for $M1$ when back to the window of opportunity for that goal at $(4, 7)$. Hence, these configurations deliver convenient information.

As explained, in order to create stable results, I ran each game numerous times with random configurations. I concluded that *cautious* and *model agent* provide $\varepsilon = 1.0$ for any environment configuration because they respect the information about window of opportunity to coordinate the execution. However, the effectiveness of the *bold agent* varies depending on both the rate of world change and relevance of the modifications.

Figure 6.5 presents the testing results with a wide range of variation of the degree of dynamism (DD). The graph replicates the tests presented in Kinny and Georgeff [1991], where the degree of relevance (DR) is 1.0, meaning that only relevant changes occur on the environments.

The results highlight the effect of the rate of world change on the effectiveness of the *bold agent*. If $DD = 0$, then *bold agents* present perfect results. In addition, this configuration behaves reasonably well in less dynamic environments where $DD \leq 0.01$. However, its performance drops exponentially as DD increases reflecting its inability to cope with dynamic environments. It presents very poor performance in configurations typical of mobile services, i.e. $0.3 \leq DD \leq 0.6$, making it unfeasible for these applications. As mentioned, *cautious* and *model agent* provide $\varepsilon = 1.0$ in any configuration.

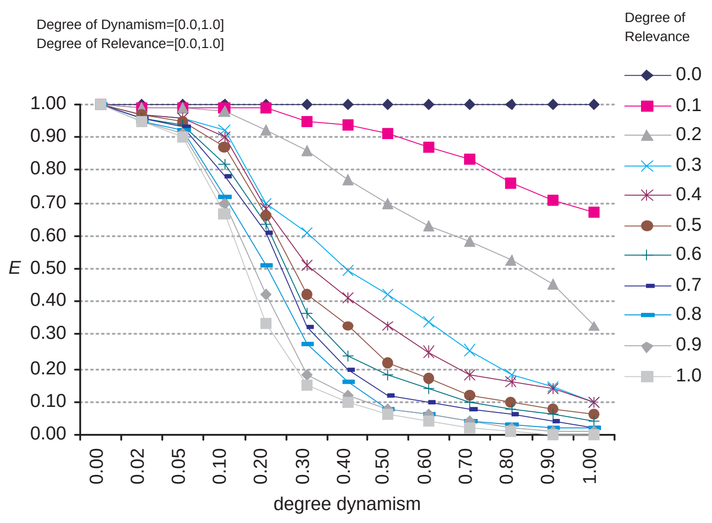


Figure 6.6: Effect of Degree of Dynamism and Relevance

Figure 6.6 presents the results for different degree of relevance (DR). Again, *cautious* and *model agent* provide perfect results. However, the effectiveness of *bold agents* is reversely proportional to the variation of DR. This can be expressed as the smaller the relevance in belief base updates (for the reconsideration of current processing), the less likely some change will affect an element that impacts the processing result. Consequently, an environment where nothing relevant is changing (i.e. $DR = 0$) behaves as a static environment.

As mentioned, neither $DR = 1.0$ nor $DR = 0.0$ reflect real world conditions. Instead, the mobile services, as an implementation of context-aware applications, represent a large amount of information about the surrounding environment and only a percentage of this knowledge gets update frequently. I proposed that $0.3 \leq DR \leq 0.6$ is representative of mobile services. However, even at these ranges, $0.3 \leq DD \leq 0.6$ bold agents present poor performance, as it can be inferred from the results.

I conclude that bold agents are not a solution for mobile services and both cautious and model agents provide deliver results. The question is: *what is the difference between using cautious and model agents*? To answer this question, next I analyse the computational performance delivered by these configurations.

6.3.2 Computational Performance

From the experiments above, it became clear that *bold agents* are not a real option for mobile services and *cautious* and *model agents* provide similar effec-

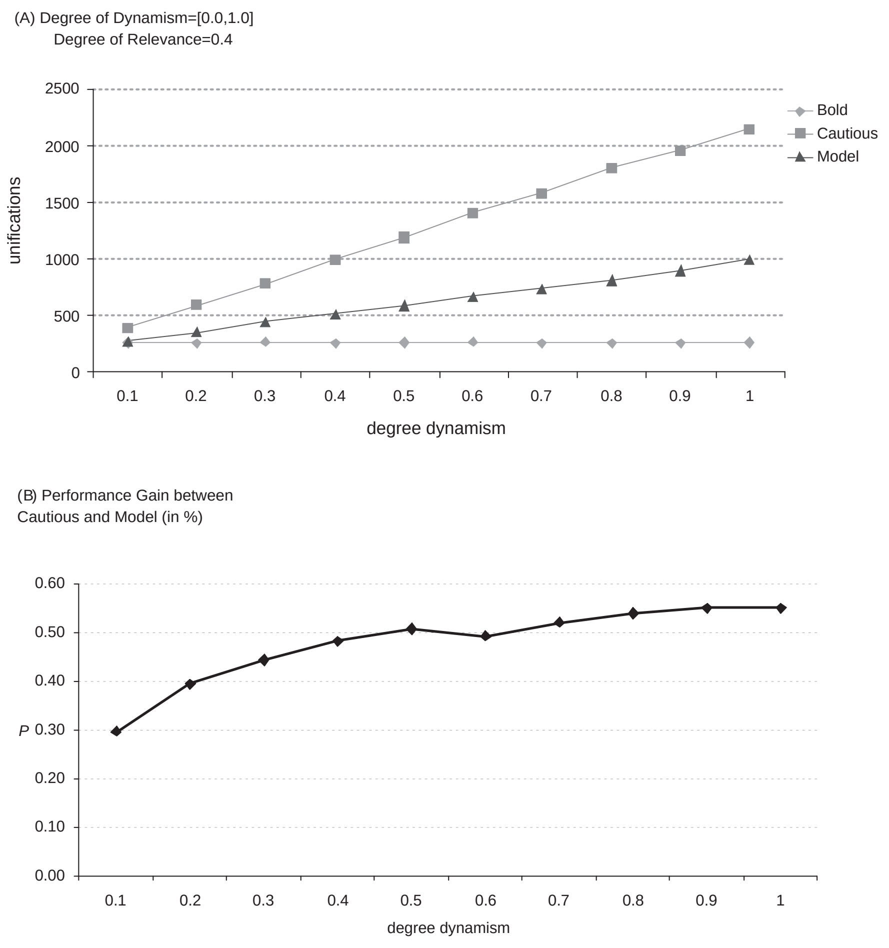


Figure 6.7: Computational Performance *versus* Degree of Dynamism

tiveness. In this second set of experiments, I focus on comparing the different in computational performance between these two reconsideration strategies when playing the scenarios with different degrees of dynamism and relevance.

I conclude that *model agents* provide a better solution for environments whose degree of relevance (DR) is representative for mobile services, i.e. $0.3 \leq DR \leq 0.6$. This result is somewhat intuitive as the model agent is able to process only relevant changes, saving computational resources. As demonstrated below, the benefit is proportional to the degrees of dynamism and relevance, as well as the complexity of the application, characterised by the size of the belief base, number of intentions, and number of plan rules.

Computational Performance Results

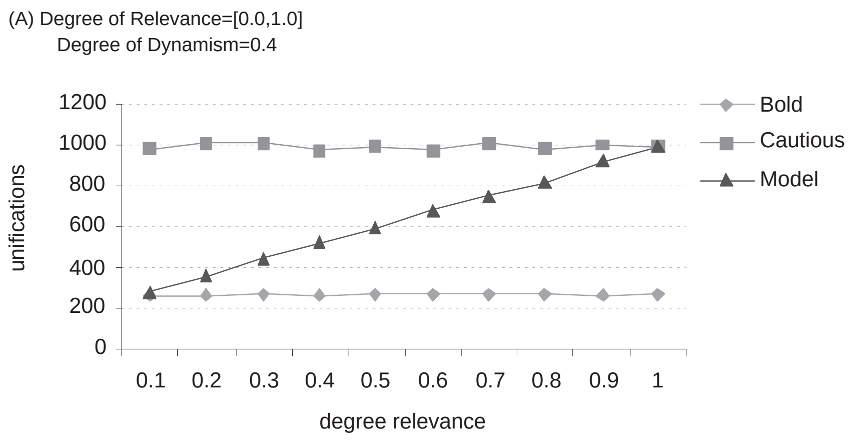
First, I analyse the computational performance results for the experiments from the testing environment above. I reiterate that I measure this parameter based on the number of unifications required to execute the game. Thus, I collected the values for the multiple repetition while running the experiments above and plotted the performance graphs.

Figure 6.7(A) provides the absolute number of unifications required (in average) to resolve the game by the three agent variations and $0.0 \leq DD \leq 1.0$. The *bold agent* presents the best performance, and that does not vary with the variation of DR. This performance behaviour can be explained as these agents are not actually observing the surrounding changes, so their environment updates are not being taken in consideration of the processing. On the other hand, the cautious agents present the worst computational performance, with resource utilisation growing proportionally to the degree of dynamism. This can be explained as cautious agents are calculating the conditions of the existing elements for every environmental update, whether it was relevant or not. Finally, the *model agent* presents a performance in between the other two, as it is able to filter the relevant updates before processing the elements' conditions.

Figure 6.7(B) presents the relative performance gain between the *cautious* and *model agents.* I suggest that it is not relevant to analyse against the performance of *bold agents*, as this configuration differs in effectiveness level. As it can be inferred, the advantage grows proportionally to the degree of dynamism of the environment, which happens as the *cautious agent* curve in Figure 6.7(A) is more steep than the one for *model agent.* This variation is influenced by the degree of relevance (DR) parameters, which is configure as $DR = 0.4$ for these scenarios.

The influence of the degree of relevance parameter is also analysed. Figure 6.8(A) presents the absolute number of unifications required to resolve the games for $0.0 \leq DD \leq 1.0$. I noticed the the values for the *bold* and *cautious agents* remain nearly static[1]. This happens because these configurations are not taking relevance in consideration. On the other hand, the number of unifications for the *model agent* increases proportionally to the degree of relevance. That is, the more relevant the environmental updates, the more likely they will affect the elements' conditions. It goes to a point where if $DR = 1.0$, then the model agent has the same performance as the cautious agents, as both have to compute the elements' conditions for every environmental update. Figure

[1]*Nearly static*: the small fluctuations happen due to variations in the environment configuration between the tests.



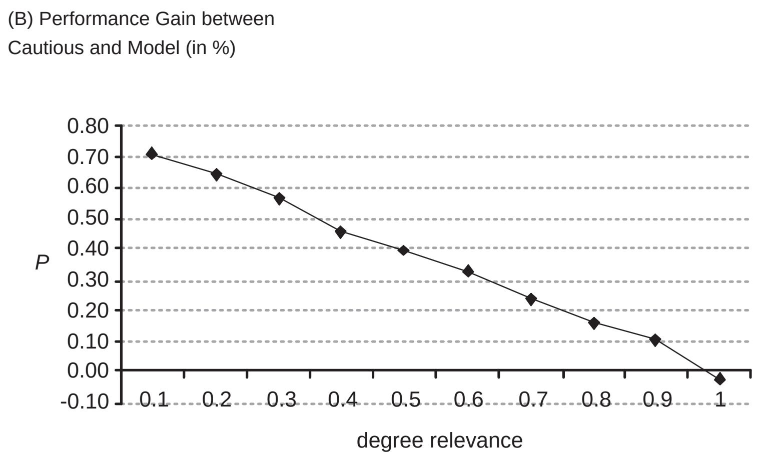


Figure 6.8: Computational Performance *versus* Degree of Relevance

6.8(B) presents the relative performance gain between the *cautious* and *model agents* for variations in degree of relevance.

Scalability

Finally, the graphs in Figure 6.9 compares the results by varying both the degree of dynamism (DD) and degree of relevance (DR) in the same configuration. Figure 6.9(A) presents the results for the illustrative scenario above. Figure 6.9(B) presents a second view of the same scenario where there are 6 goals configured in the game. This is equivalent of having a scenario where the user is heading to a meeting and other 5 sideline meetings pop up one after the other. The scenario is based on the *coordination in emergency response*, popular in the multi-agent system literature[2], *viz.* Schurr et al. [2006]. The parameters for each simulation are also described.

The main difference is the reduction in performance gains between the scenarios. The model agent provides maximum performance gain of 70% for the scenario with 6 goals and almost 80% for the scenario with 2 goals. This difference can be explained as more complex games require more cycles to resolve the application's main goal (i.e. retrieve information), which reduces the weight of the reconsideration operations. Hence, the reduction in relative performance is related to the complexity of the game and not a characteristic of the reconsideration strategy *per se*.

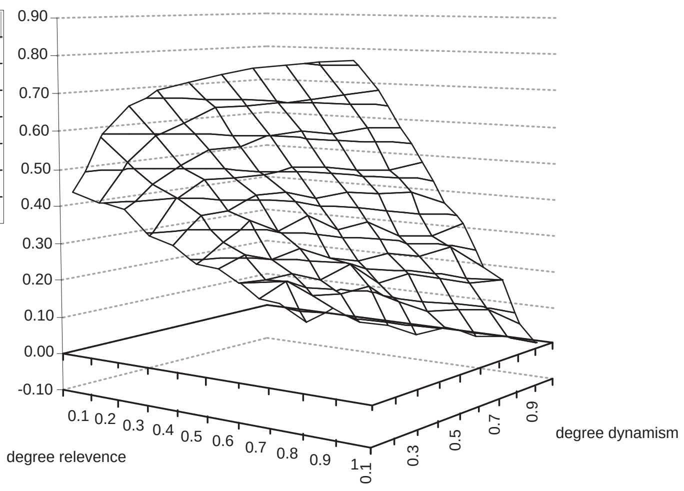
Moreover, the game with more goals starts with a lower performance that increases more steadily, compared to the simpler game. For example, if the configuration has $DD = 0.1$ and $DR = 0.1$ (i.e. almost static), the gain in performance for (A) starts in around 40%, whereas the gain in performance for (B) starts in around 30%. The explanation is the same as above: more complex games require more cycles to resolve the application's main goal, which reduces the weight of the reconsideration operations, specially in the case where there are less elements to reconsider.

Therefore, I conclude that scalability factors, such as the increase of game complexity do not affect the relative gain in performance of the proposed model over the cautious agent configuration. The gains in performance seem to be solely related to differences in environment configuration, namely variations in the degree of dynamism and relevance environmental updates.

[2] *Emergency Response Systems*: this scenario is characterised by being multiplayer, highly dynamic, simultaneous events, and multiple goals happening in parallel. Although this scenario highlights issues of multiplayer and collaboration, I stress that the focus is on analysing the application's performance.

(A)

PARAMETER	VALUE
NR_START_GOAL	1
NR_END_GOAL	2
SIZE_WOO	2
PRIORITY_INCREASE	1
SIZE_BELIEFBASE	32
DEGREE_DYN	[0.0,1.0]
DEGREE_REL	[0.0,1.0]



(B)

PARAMETER	VALUE
NR_START_GOAL	1
NR_END_GOAL	6
SIZE_WOO	2
PRIORITY_INCREASE	1
SIZE_BELIEFBASE	32
DEGREE_DYN	[0.0,1.0]
DEGREE_REL	[0.0,1.0]

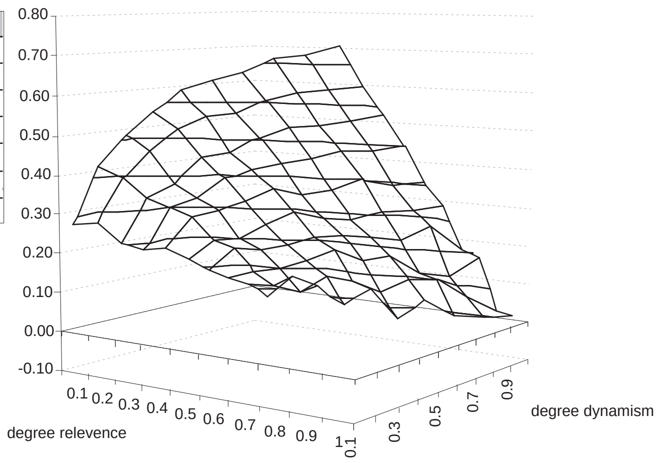


Figure 6.9: Scalability

6.4 Conclusion

In this chapter, I constructed a highly parameterised simulation environment by combining the prototype application from Chapter 5 with the MobileWorld simulation environment. The combined system allows the dynamic characteristics of the agent and environment to be varied over a wide range and enables comparison over a large space of agent/environment combinations. Using this testing environment, I was able to measure and compare the agent effectiveness in a simulation environment when implementing the proposed model. Moreover, I compare the properties of different strategies for reacting to environment changes.

The results underline the gain in computation performance provided by the proposed model. I demonstrated that the model agent provides similar effectiveness than the cautious agents, which is the configuration used by several field agent implementations. However, the model agent provides significant computational performance advantage, by being able to exploit relevance of the environment updates for the reconsideration strategy.

This optimisation means that less CPU-cycles are required to run the application, which reflects in the application's overall performance and reduction of battery utilisation. Hence, I conclude that the proposed model is a better option to implement mobile services than other solutions in the field.

Chapter 7

Conclusion

"Begin at the beginning ... then go on till you come to the end: then stop'."
Lewis Carroll, Alice's Adventures in Wonderland.

The objective of this work was to outline an extended computational model to support the development of intelligent mobile services. In Chapter 1, I stated that three key requirements for developing these services are:

- (i) to support the application's interactions, enabling the application to work on behalf of the user as autonomously as possible;
- (ii) to provide quality information, through a flexible, adaptive inference system capable of adjusting the processing line in response to internal and external conditions, and;
- (iii) to be resource aware, considering the limitations of mobile computing.

I argued that solutions based on reactive decision loops, such as applications developed in common programming languages like C++ and Java, are unfit to implement these applications. They result in pure reactive systems that, although capable of effective behaviour in some dynamic environments, cannot guarantee the performance in unanticipated situations.

Moreover, I alluded that the fields of context-awareness and agent-based computing offer an alternative. In this scope, I focused on exploiting a specific class of agents for the solution: the *Belief-Desire-Intention* (BDI) model. This paradigm provides several desirable features, such as structures for knowledge representation, supporting proactiveness, responsiveness, and situatedness, introduced in Wooldridge and Jennings [1995] and Jennings et al. [1998].

However, while analysing the requirements to implement intelligent mobile services using BDI-model agents, I discovered that current solutions lack the reactiveness required to support highly dynamic environments. Although initial developments were inspired by reactive approaches, BDI-model evolution focused on providing adaptive behaviour, to the detriment of reactiveness. Current designs use deliberation cycles that implement a "check-deliberate-action" loop, which implies applications overcommitted to the planning phase. This deficiency impacts the requirements for reacting to the environment, which compromises the support for (ii) providing quality information and (iii) resource awareness. Moreover, it restricts its applicability for applications that operate in highly dynamic environments. Consequently, I argued that current implementations cannot be used "as is" to develop intelligent mobile services.

Therefore, my research revolved around the following research question:

> What is the architecture of the inference system needed to support the development of intelligent mobile services?

Throughout this work, I developed an extended BDI-model design that equips the inference system with additional operational and knowledge structures. This new design supports parallel planning, enhanced adaptiveness, impromptu reaction to environmental changes, and better balance between reactiveness and proactiveness in the deliberation behaviour. The model is composed of:

1. the *context observer module* that observes changes of the environment and rationalises relevant events that signal the planning module when to reconsider an intention's processing thread;

2. the *planning module* that includes enhanced operation structures of planning threads, plan selection, plan revision, and basic action execution, and;

3. the *learning module* that provides the structures to plug-in external functionality to adjust the configurable parameters and knowledge.

The new operations structures introduced in the proposed model support parallel planning through planning threads that can be started, stopped, paused, and resumed independently. This feature is required to provision applications capable to handle multiple tasks simultaneously, a pre-requisite for mobile services. In addition, the model adds the knowledge structures to represent (a) prioritisation, conflict resolution, and cost rules, and; (b) the filter of relevant events, to rationalise the environmental observation process.

The first group must be programmed during the development phase. Relevance filters are inferred by the application from the current situation and in-processing elements. These structures address the issues of (ii) providing quality information and (iii) resource awareness, as demonstrated by the case study scenarios presented in Chapter 6.

As a limitation, this approach requires the explicit representation of the extra knowledge – i.e. cost rules, prioritisation rules, conflict rules, proximity conditions, etc. One can argue that this requirement has two side-effects: first, it augments the complexity of development process and, consequently, the probability of errors, and; second, it requires more computation resources (e.g. processor, memory) to operate, thus jeopardising its applicability in devices with constraint resources. I acknowledge these claims and suggest that this issue can be partially solved through the creation of "generic rules" that cover different scenarios with similar conditions. Moreover, in Chapter 6, I demonstrated that the proposed model actually improves the resource utilisation when operating in highly dynamic environments.

In addition, the model provides the structures to exploit environmental events to coordinate the deliberation process. It introduces the notion of "window of opportunity", which describes an area where the application can provide perceivably convenient information with regards to a task. In order to support this feature, the model provides: (a) the function to calculate proximity conditions; (b) the knowledge structures to represented relevant filters, and; (c) the operations to coordinate the deliberation process. The resulting solution can determine *when* to reconsider the current deliberations and *what* to do – i.e. to start, stop, pause, or resume the related processing thread. I argue this feature simplifies the coding process and enhances overall application performance, boosting reactiveness to external events.

The proposed model can also be applied in different knowledge domains that involve high-dynamic environments. In these different applications, the main difference is that the context observer must be equipped to realise and make sense of different dimensions of context information where they are immersed – e.g. time, location, objects present, activity, social setting, environment, and culture. Moreover, the function to calculate proximity must be able to infer the in-out condition in relation to the window of opportunity for these other dimensions. The prototype implementation introduced provides generic support re-usable in a wide range of knowledge domains – e.g. mobile services, robotics, etc – requiring the customisations of the applications' interface to the external world.

Revisiting the Requirements Analysis

Chapter 2 provided an overview of the related techniques and theories in order to distil the fundamental concepts behind this problem environment. It derived a list of requirements to support the development of intelligent mobile services. Although many are fulfilled to varying degrees by existing solutions, some require an extension of the current BDI-model. From that point, my research focused on what aspects of current BDI-model technology could be re-used and what needed to be extended. Table 7.1 summarises the list of requirements introduced in Chapter 2 and extends the analysis presented in Section 2.4. The next paragraphs detail the entries.

First, the application must provide *enhanced user experience*, that is the structures to support directability, personalisation, teachability, and transparency. It aims to improve how the solution is perceived, learned and used. The BDI-model supports these requirements, having innate features for: *directability*, providing a method to explain the decision process based on the deliberation steps; *personalisation*, providing the knowledge structures capable of modelling the user's preferences and behaviour; *teachability*, allowing new knowledge (beliefs, desires, plan rules) to be asserted during run time, and *transparency*, explaining the decision taking process based on executed rules. Being an extension of the BDI-model, the proposed model inherently embodies these characteristics. Moreover, it adds the possibility of explaining *how* environmental events influenced the deliberation process, which possibly adds value to the final solution.

Second, the application must be *proactive*, that is it must be able to consider the possibility of future events and take actions on behalf of the user without requiring a direct command. The BDI-model inherently supports this item by implementing goal-oriented behaviour. Applications developed from this model are able to receive a goal from the user and compose its own plans for achieving that goal. The application's purposeful behaviour, or commitment, provides a level of stability to the reasoning process, and helps to provide coherent, quality information. In practice, this is done in the 2APL-M prototype implementation, by programming mobile applications using goals and plan rules. Moreover, the proposed model introduces features to improve the balance between pro-active and reactive behaviour, which help to leverage this feature without compromising the application's overall performance.

The application must also be *responsive*; able to adapt its processing in reaction to environmental events. Although BDI-model implementations provide some level of reactiveness in the deliberation process, current designs are based on the "check-deliberate-action" loop, where the application is commit-

Requirement	Solution provided by the BDI-Model	Extended solution provided by proposed model
Enhanced user interface incl. (i) Directability, (ii) Personalisation, (iii) Teachability, and (iv) Transparency	Decision process can be explained based on deliberation steps; allows to model user's preferences and behaviour ; provides tracing information about decision taking process	Inherits decision capability from the BDI-model; adds the possibility of explaining "how" environmental events influenced the deliberation process
(v) Pro-Activeness	Supported by goal oriented behaviour	Inherits goal-oriented behaviour from the BDI-model; adds features to promote better balance between pro-active and reactive behaviour ; exploits the notion of window of opportunity to enhance reactiveness and deliberation process coordination
(vi) Responsiveness	Supported by allowing reconsideration of environmental situation in between plan steps	Enhanced responsiveness by providing the operational and knowledge structures to react to environmental events, adapting the deliberation behaviour
(vii) Means-end coherence	Goal-oriented behaviour and desire-belief plan selection	Inherits features from the BDI-model
(viii) Adjustable and self-configurable	Allows adjusting of current processing response to external in events	Enhanced deliberative behaviour by introducing the operational and knowledge structures to react to environmental events and adjust current deliberations by pausing, stopping, and resuming existing planning threads; introduces the concept of learning module to provide structures to adjust configurable parameters and knowledge on-demand, during run-time
(ix) Structures for knowledge representation	Beliefs, Desires, Intentions, and Plan Rules	Inherits structures from BDI-model; adds knowledge structures to represent (a) prioritisation , conflict resolution, and cost rules, and; (b) filter of relevant events, to rationalise the environmental observation process
(x) Situatedness	Supports interaction and representation of local environment	Inherits structures from BDI-model; adds the context observer module to perceive and rationalise relevant changes of the environment, and secondly, with the sensor module to perceive the environment and update the belief base
(xi) Scalable	Simplified planning process	Introduces extended computational model that takes advantage of environmental events to coordinate the deliberation process; integrates the elements to observe changes of the environment and signal the deliberation cycle how to react to these events; this configuration provides offers significant gains in computational performance
Applicability	Applications that execute in dynamic environments and require a certain level of adaptability, pro-activeness, and goal oriented behaviour .	Besides the features of the BDI-model, the extended model is better suited for applications that operate in highly dynamic environments, which require continuous verification of the operation conditions to ensure coherence, such as mobile services, robotics, large scale social simulation systems, controlling systems for highly dynamic operations, and others.

Table 7.1: Analysis of the Solutions Provided by the Proposed Model

ted to complete the current planning phase before considering changes within the environment. This architecture compromises supporting highly dynamic environments, where higher degrees of dynamism affect the performance and quality of information provided. The proposed model corrects this limitation by including a *context observer module*, which observes environmental changes and signals the planning module when intention reconsideration is required. This feature enhances application responsiveness to environmental events, leading to applications better suited to the targeted environment.

The application must be able to sustain *means-end coherence*; allowing it to work autonomously and behave flexibly in reaction to unexpected situations. This feature leads to applications that are stable and provide coherent, quality information. The BDI-model supports this requirement by implementing goal-oriented behaviour and plan selection. In practice, 2APL-M implements this feature based on the concept of "on-demand plan selection" where the application selects the plan to follow based on desire-belief reasons. Hence, the proposed model inherently provides this characteristic, being an extension of the BDI-model fulfilling this requirement.

It must also be *adjustable* and *self-configurable*; able to learn from the environment and experience, and adjust the deliberation behaviour. The BDI-model partially supports this requirement by allowing deliberation reconsideration during processing. In practice, 2APL-M implements this feature via plan revision rules. The proposed model extends this support by introducing the operational and knowledge structures to react to environmental events and adjust current deliberations by pausing, stopping, and resuming existing planning threads. In addition, it introduces the *learning module*, which provides the structures to plug-in external functionality that adjusts the configurable parameters and knowledge.

Moreover, the application must provide the *structures for knowledge representation* to represent internal and external status. The BDI-model provides data representation that support this requirements, namely: *beliefs*, which represent the state of the environment, such as environmental variables, user's profile and activities; *desires*, which outline the application's behaviour, that is the generic goals that the application should pursue when proper conditions occur; *intentions*, which represent the goals to which the application is committed, and *plans*, which represent the practical reasoning rules on *how* agents infer *what to do* to achieve a goal. The 2APL-M implementation provides these software application data structures that hold the knowledge representation. The proposed model also adds the knowledge structures to represent (a) prioritisation, conflict resolution, and cost rules, and; (b) filter of relevant events, to rationalise the environmental observation process. These structures help to resolve the issues of (ii) providing quality information and (iii) resource awareness by improving the deliberation behaviour.

The application must be *situated*, providing the structures to represent and reason about the surrounding environment. As mentioned above, the BDI-model provides structures for knowledge representation. In the proposed model, techniques from the context-awareness field provide enhanced support: first, by introducing the *context observer module* to perceive and rationalise relevant changes of the environment, and secondly, with the *sensor module* to perceive the environment and update the belief base. Agent technology inherently provides the enhanced deliberation systems and pro-active behaviour required to make sense of the collected information. Hence, the combination of BDI-model and context-context aware computing elements provides extended support to implement aspects of situatedness in mobile services, fulfilling this requirement.

Finally, the application must be *scalable*. This is a key requirement for applications that operate in resource-constrained environments, such as mobile services. The BDI-model provides some features for system scalability related to practical reasoning. The practical reasoning deliberation model focuses on a single line of reasoning: interleaving plan selection, formation, and execution. In short, the system forms a partial overall plan, determines and executes a means of accomplishing the first sub-goal of the plan, then repeatedly expands and executes the near-term plan of action until the goal is reached or abandoned.

However, this feature alone does not provide a solution for the problem of scalability in mobile services. As mobile services must handle multiple activities simultaneously, due to the nature of the environment, it implies that the planning system must be able to process parallel planning threads, which requires potentially large resource allocations. Further, the "check-deliberate-action" loop proposed by the BDI-model compromises processing performance and information quality when operating in highly dynamic environments. I demonstrated these limitations through the case studies in Chapter 6.

The proposed model provides the extended computational model that takes advantage of environmental events to coordinate the deliberation process, as presented in Chapter 3. The model integrates the elements to observe changes of the environment and signal the deliberation cycle how to react to these events. The model is also equipped to handle the issues of current intentions, parallel planning, conflicts, prioritisation, reconsideration and adaptation, which is a pre-requisite to support the development of mobile services.

Chapter 4 outlined the design model for the inference system, proposing practical solutions for the pending issues of the conceptual model. Chapter 5 introduced a prototype implementation of the proposed model, proving it is feasible as a software application. Finally, Chapter 6 considered the model's effectiveness in a simulated environment. The results highlighted the proposed model's gain in computational performance over other agent configurations, based on how they implement the reconsideration process, i.e. *bold agents* that never consider the impact of environmental changes on the current processing, and *cautious agents* that consider the impact at each deliberation step.

I demonstrated that the proposed model not only provides similar effectiveness to *cautious agents*, which is the configuration used by several agent implementations in the field, but also offers significant gain in computational performance. Hence, I argued that the proposed model provides better scalability than other solutions, fulfilling the requirement for scalability.

Consequently, as represented in Table 7.1, the proposed model is better suited for the development of applications that operate in highly dynamic environments. For example, solutions that require continuous verification of the operation conditions to ensure coherence, such as mobile services, robotics, large scale social simulation systems, controlling systems for highly dynamic operations, and others.

Contribution to Agent Research

The proposed model contributes to existing agent models, from concepts regarding the process of providing cognitive behaviour in mobile services adding value to the existing BDI-model agent languages implementations, and addressing the fundamental issues of balancing pro-activeness and reactiveness.

There are several contributions of the proposed model to the field of agent research. As well as desired features that were reused for this development. For example, it re-used concepts from cognitive computing systems, such as SOAR, introduced in Rosenbloom et al. [1993], and CLARION, presented in Sun [2008], separating knowledge representation and planning and goal oriented behaviour and rule-based inference systems. Hence, it allows the application to behave in a goal-oriented manner, provide rule-based reasoning systems that adapt the execution path in response to changes of the environment, represent the environment by labelling its elements, and learn from the environment and experience.

The proposed model extends concepts from practical reasoning model, like *Procedural Reasoning System* (PRS), introduced in Georgeff [1987]. In this model, the planning system interleaves plan selection, formation, and execution and will try to fulfil any goals and intentions it has previously decided upon unless some new fact or request activates a new plan, at which time PRS will reassess its goals and intentions. However, the reassessment will occur only after completing the current planning step. In the proposed model, the reassessment procedure is event-based, activated by environmental conditions, which allows faster interruption and reconsideration of the current process, improving the application's reactiveness to environmental events.

Consequently, the proposed model provides a new approach to allow agent-based applications to exploit features of the environment to infer *when* to revise current processing lines. The BDI-model is extended with additional operational and knowledge structures that allow parallel planning, enhanced adaptiveness, impromptu reaction to environmental changes, and to promote better balance between reactiveness and proactiveness for the deliberation behaviour. I argue that applications developed using this model are better equipped to cope with environmental instabilities yet optimised for resource utilisation.

Moreover, the proposed model's reconsideration strategy falls between the agent configurations proposed in Kinny and Georgeff [1991], which are the basis for current implementations; that is *bold agents* that never consider the impact of environmental changes on the current processing, and *cautious agents*, which consider the impact at each deliberation step. The results presented in Chapter 6 demonstrated that the model agent provides similar effectiveness to cautious agents with a significant computational performance advantage.

Comparatively, Schut et al. [2004] analyses the performance of the different reconsideration strategies from the view of one agent action in a simulation model. That work considers two different models of intention reconsideration, based on discrete deliberation scheduling, which analysis in terms of conventional decision theoretic models of optimal action, and; partially observable *Markov decision process*, whose solution implies in finding an optimal intention reconsideration policy. I acknowledge that the proposed model can be extended to produce results equivalent to the ones reached in that work. However, I took the alternative approach by reducing the scope of the analysis for the purpose of this work. This work's approach leads to non-intuitive testing environments, whose results can not directly relate to real world situations. The simulation environment introduced in Chapter 6 produces results that are closer to real world environmental conditions; consequently, it is more intuitive.

Finally, the proposed model offers an alternative to existing inference systems like AgentSpeak(XL), introduced in Bordini et al. [2002], JACK, introduced in Padgham and Winikoff [2002], and 3APL, introduced in Hindriks [2001]. In these implementations, the deliberation cycle follows the "check-deliberate-action" loop structure, previously described in Rao and Georgeff [1995], where the application is committed to the current deliberation and, consequently, is inflexible. The model extends this concept by allowing interruption of the loop based on relevant environmental events. Consequently, the application reacts faster to environmental conditions with no impact on its pro-active features. This is desirable for applications that operate in highly dynamic environments, like mobile services.

I concluded that integrating the proposed model with an existing agent language offers numerous advantages, such as:

- it enhances the reactiveness to environmental changes in a controlled, rationalised way;
- the context observer provides the solution to determine *when* to reconsider the processing without having to incorporate special structures on the plans' conditions, and;
- it rationalises the reconsideration process by implementing filters for the relevant (i.e. potentially impacting) environmental changes.

I argue that these structures add value to the existing platforms by providing a solution to the fundamental problem of balancing reactiveness and pro-activeness.

Contribution to Intelligent Mobile Services

I conclude that by fulfilling these requirements the proposed model addresses the three key requirements to develop mobile services, proposed in Chapter 1.

Firstly, it utilises the inherent BDI-model features of proactiveness, means-end coherence and situatedness to provide the elements to (i) *support the application's interactions.* It allows building applications that are able to consider the possibility of future events and take actions; working autonomously on behalf of the user without requiring a direct command to execute an action, responding flexibly to unexpected situations, and representing and processing information about the surrounding environment.

Secondly, it balances pro-activeness and the model's inherent reactiveness and ability to exploit environmental events and support the ability to (ii) *provide quality information.* Applications implemented using the proposed model are equipped to deliver information within the window of opportunity regardless of the environment's dynamism, as demonstrated by the case studies in Chapter 6. This behaviour guarantees convenience and coherence by the information system.

Finally, applications implemented using the proposed model are inherently (iii) *resource aware*, due to the gains in computation performance provided by the enhanced deliberative behaviour, resulting from the ability to exploit relevance of the environment updates for the reconsideration strategy. This optimisation means that less CPU-cycles are required to run the application in similar circumstances, which reflects in the application's overall performance and reduction of battery utilisation.

Therefore, I conclude that the architecture outlined by the proposed model answers the research question. It fulfils all requirements for developing intelligent mobile services and provides a number of contributions to the literature. The proposed model can be applied to implement applications in different knowledge domains where software applications must operate in a dynamic environment, use context information for decision making, and implement both reactive and proactive behaviour. Furthermore, Different agent platforms could be adapted to support the proposed design to reduce CPU cycles and increase overall application performance. Hence, it provides gains in qualitative and quantitative performance when compared to other agent models.

Future Work

There is considerable scope for possible future work related to the activities developed throughout this thesis:

In Chapter 3, I introduced a function that calculates the costs of selected plans on-demand using the cost rules represent in the belief base. However, in the BDI-model, the planning process behaves following the *think-act* model, where the whole plan ahead of execution is not formed. Hence, it is not trivial to apply this recursive formula to calculate the plan cost as the cost element cannot be quantified immediately. I introduced a simplistic solution in Chapter 5, based on a programming construct; however, this solution is not ideal for a real world application, as it requires the programmer to observe some restrictions, such as having only one term to avoid inconsistencies. As future research, sought a generic solution for calculating plan costs that is more compatible with the BDI-model deliberation process. This will improve support for realistic applications that need to decide between multiple options during operations.

Exceptions during basic action execution trigger the learning process defined in Chapter 3. As future work, this module can be further extended towards pro-active learning; that is, learning by observing the environmental conditions and adjusting the current parameters during operations.

In Chapter 4, I alluded that different agent platforms can be adapted to support the elements of the proposed model. As future work, I proposed to apply the design model to extend other agent languages, exploring the effectiveness of the model as a generic solution to improve agent technology.

In Chapter 5, I proposed several reductions to facilitate the proof-of-concept implementation. For example, the functionality to calculate proximity – part of the context observer module – is defined in terms of time-space conditions. As future work, the context observer can take advantage of other dimensions of context information. This would allow the implementation of applications in different knowledge domains.

Finally, a future work would be to extend the pilot application to support the development of "real world" applications. The prototype's core functionality – i.e. knowledge bases and deliberation structures – is currently fully operational. However, in order to create larger, more integrated solutions, it must be extended with field technologies. This is a problem of software engineering towards distribution-ready intelligent mobile services.

Bibliography

G. Abowd, C. Atkeson, J. Hong, S. Long, R. Kooper, and M. PinkertonSA. Cyberguide: A mobile context-aware tour guide. *ACM Wireless Networks*, 3:421–433, 1997.

J. R. Anderson and K. A. Gluck. *Cognition & Instruction: Twenty-five years of progres*, chapter What role do cognitive architectures play in intelligent tutoring systems?, pages 227–262. Erlbaum, 2001.

J. R. Anderson and C. Lebiere. *The Atomic Components of Thought*. Lawrence Erlbaum, 1998.

M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context-aware systems. *International Journal Ad Hoc Ubiquitous Computing*, 2(4):263–277, 2007.

C. Bartl and D. Doerner. Comparing the behavior of psi with human behavior in the biolab game. In F. Ritter and R. M. Young, editors, *Proceedings of the Second International Conference on Cognitive Modeling*. Nottingham University Press, 1998.

M. Beer, M. d'Inverno, M. Luck, N. Jennings, C. Preist, and M. Schroeder. Negotiation in multi-agent systems. *Knowledge Engineering Review 14*, pages 285–289, 1999.

F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. JADE - A Java Agent Development Framework. In *Multi-Agent Programming*, pages 125–147. Kluwer, 2005.

F. Bergenti, A. Poggi, B. Burg, and G. Claire. Deploying FIPA-compliant systems on handheld devices. *IEEE Internet Computing*, 5(4):20–25, 2001. ISSN 1089-7801.

P. Berry, B. Peintner, K. Conley, M. Gervasio, T. Uribe, and N. Yorke-Smith. Deploying a personalized time management agent. In *Proceedings of the fifth international joint conference on Autonomous Agents and*

Multiagent Systems, pages 1564–1571, New York, NY, USA, 2006. ACM. ISBN 1-59593-303-4.

R. H. Bordini, A. L. C. Bazzan, R. O. Jannone, D. M. Basso, R. M. Vicari, and V. R. Lesser. AgentSpeak(XL): Efficient intention selection in BDI agents via decision-theoretic task scheduling. In C. Castelfranchi and W. L. Johnson, editors, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS-2002)*, pages 1294–1302, New York, USA, 2002. ACM Press.

R. H. Bordini, J. F. Hubner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason.* Wiley-Interscience, 2007.

M. E. Bratman. *Intentions, Plans, and Practical Reasoning.* Harvard University Press, Cambridge MA, USA, 1987. ISBN 1575861925.

L. Braubach, A. Pokahr, and W. Lamersdorf. JADEX: A short overview. In *Proceedings of Main Conference Net.ObjectDays 2004*, 2004.

P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. Jack Intelligent Agents - components for intelligent agents in JAVA, 1999.

H. Chalupsky, Y. Gil, C. A. Knoblock, K. Lerman, J. Oh, D. V. Pynadath, T. A. Russ, and M. Tambe. Electric elves: Applying agent technology to support human organizations. In H. Hirsh and S. Chien, editors, *Proceedings of the 13th International Conference of Innovative Application of Artificial Intelligence (IAAI-2001).* AAAI Press, 2001.

S. F. Chipman and A. L. Meyrowitz. *Foundations of Knowledge Acquisition: Machine Learning.* Kluwer Academic Publishers, 1993.

P. R. Cohen and H. J. Levesque. Intention is Choice with Commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

M. Dastani, F. Dignum, and J.-J. Meyer. Autonomy and Agent Deliberation. In M. Rovatsos and M. Nickles, editors, *Proceedings of the First International Workshop on Computatinal Autonomy - Potential, Risks, Solutions (Autonomous 2003)*, pages 23–35, Melbourne, Australia, July 2003.

M. Dastani, M. van Birna Riemsdijk, and J.-J. C. Meyer15. *Programming Multi-Agent Systems in 3APL*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter 2, pages 39–67. Springer US, May 2006.

M. Dastani, D. Hobo, and J.-J. Meyer. Practical Extensions in Agent Programming Languages. In A. Press, editor, *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07)*, 2007.

F. de Boer, R. M. van Eijk, W. van der Hoek, and J.-J. Meyer. A fully abstract model for the exchange of information in multi-agent systems. *Theoretical Computer Science*, 290(3):1753–1773, 2003.

A. K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology, November 2000.

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

J. Doyle and R. H. Thomason. Background to Qualitative Decision Theory. *AI Magazine*, 20(2):55–68, 1999.

E. H. Durfee. Practically coordinating. *Artificial Intelligence Magazine*, 20(1):99–116, Spring 1999.

E. A. Emerson. *Handbook of theoretical computer science (vol. B): formal models and semantics*, chapter Temporal and modal logic, pages 995–1072. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-444-88074-7.

M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges: data-centric networking for invisible computing: the portolano project at the university of washington. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 256–262. ACM Press, 1999. ISBN 1-58113-142-9.

R. Fikes and N. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

M. P. Georgeff. Reactive Reasoning and Planning. In *Proceedings AAAI-87*, pages 677–682. American Association of Artificial Intelligence, 1987.

M. P. Georgeff, A. L. Lansky, and M. Schoppers. Reasoning and Planning in Dynamic Domains: An Experiment with a Mobile Robot. Technical Report Technical Note 380, Artificial Intelligence Center, SRI International, Menlo Park, CA, 1987.

M. Ghallab, D. S. Nau, and P. Traverso. *Automated planning: Theory and Practice.* Morgan Kaufmann Publishers, May 2004.

F. Gobet and P. C. R. Lane. CHREST, a tool for teaching cognitive modeling. In *ICCM*, 2004.

J. Goguen. Algebraic techniques. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 1, pages 217–261. Academic Press, New York, 1989.

C. Graham and J. Kjeldskov. Indexical representations for context-aware mobile devices. In *Proceedings of The IADIS e-Society Conference*, pages 373–380, Lisbon, Portugal, 2003. IADIS Press.

T. Gu, H. K. Pung, and D. Q. Zhang. A middleware for building context-aware mobile services. In *IEEE Vehicular Technology Conference (VTC-Spring 2004)*, Milan, Italy, May 2004.

K. Hindriks. *Agent Programming Languages: Programming with Mental Models.* PhD thesis, University of Utrect, February 2001.

J. I. Hong and J. A. Landay. A context/communication information agent. *Personal Ubiquitous Computing*, 5(1):78–81, 2001. ISSN 1617-4909.

A. C. Huang, B. C. Ling, and S. Ponnekanti. Pervasive computing: what is it good for? In *1st ACM international workshop on Data engineering for wireless and mobile access*, pages 84–91. ACM Press, 1999. ISBN 1-58113-175-5.

M. Inverno and M. Luck. Engineering AgentSpeak(L): A Formal Computational Model. *Journal of Logic and Computation*, 8(3):233–260, 1998.

M. Inverno and M. Luck. *Understanding Agent Systems.* Springer-Verlag, 2nd edition edition, 2003.

M. Inverno, K. Hindriks, and M. Luck. *Formal Architecture for the 3APL Agent Programming Language.* Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004.

C. L. Isbell and J. S. Pierce. An Interface-Proactivity Continuum for Adaptive Interface Design. In *Proceedings of the 11th International Conference on Human-Computer Interaction (HCII2005)*, 2005.

R. C. Jeffrey. *The Logic of Decisions.* University Of Chicago Press, 2nd edition, 1990.

N. R. Jennings and M. Wooldridge. Applications of intelligent agents. *Agent technology: foundations, applications, and markets*, pages 3–28, 1998.

N. R. Jennings, K. Sycara, and M. J. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

M. Khedr and A. Karmouch. ACAI: agent-based context-aware infrastructure for spontaneous applications. *Journal of Network and Computer Applications*, 28:19–44, 2005.

D. Kinny and M. Georgeff. Commitment and effectiveness of situated agents. In *In Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 82–88, Sydney, Australia, 1991.

F. Koch and I. Rahwan. The role of agents in intelligent mobile services. In M. W. Barley and N. Kasabov, editors, *7th Pacific Rim International Workshop on Multi-Agents, Revised Selected Papers*, volume 3371 of *Lecture Notes in Computer Science.* Springer-Verlag GmbH, 2005.

F. Koch and L. Sonenberg. Using multimedia content in intelligent mobile services. In *Proceedings of the 2nd Latin American Web Congress and the 10th Brazilian Symposium on Multimedia and the Web*, Ribeirao Preto, BR, October 2004.

F. Koch, J.-J. C. Meyer, F. Dignum, and I. Rahwan. Programming deliberative agents for mobile services: the 3APL-M Platform. In *Proceedings of AAMAS'05 Workshop on Programming Multi-Agent Systems (ProMAS'2005).*, 2005.

B. N. Kokinov. The DUAL cognitive architecture: A hybrid multi-agent approach. In *ECAI*, pages 203–207, 1994.

R. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceeding of AAAI'97*, 1997.

J. Lehman, J. E. Laird, and P. Rosenbloom. A gentle introduction to soar, an architecture for human cognition. Technical report, University of Michigan, 2006.

J. W. Lloyd. *Foundations of Logic Programming.* Spring-Verlag, 1987.

P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, 1994.

V. Mascardi, D. Demergasso, and D. Ancona. Languages for Programming BDI-style Agents: an Overview. In F. Corradini, F. D. Paoli, E. Merelli, and A. Omicini, editors, *Proceedings of WOA 2005*, pages 9–15. Pitagora Editrice Bologna, 2005. ISBN 88-371-1590-3.

E. Mendelson. *Introduction to Mathematical Logic.* D. Van Nostrand Company, 1979.

K. Myers, P. Berry, J. Blythe, K. Conley, M. Gervasio, D. McGuinness, D. Morley, A. Pfeffer, M. Pollack, and M. Tambe. An intelligent personal assistant for task and time management. *AI Magazine*, 28:47–61, 2007.

B. Nebel and J. Koehler. Plan Reuse versus Plan Generation: A Theoretical and Empirical Analysis. *Artificial Intelligence*, 76(1-2):427–454, 1995. ISSN 0004-3702.

A. Newell, J. C. Shaw, and H. A. Simon. Report on a general problem-solving program. In *IFIP Congress*, pages 256–264, 1959.

N. J. Nilsson and R. E. Fikes. STRIPS: a New Approach to the Application of Theorem Proving to Problem Solving. SRI Project 8259 43, Stanford Research Institute, Menlo Park, CA, USA, October 1970.

D. A. Norman. *The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances are the Solution.* The MIT Press, 1999.

L. Padgham and M. Winikoff. Prometheus: a methodology for developing intelligent agents. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 37–38, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-480-0.

A. Pashtan, R. Blattler, A. Heusser, and P. Scheuermann. CATIS: A context -aware tourist information system. In *Proceedings of the 4th International Workshop of Mobile Computing*, 2003.

M. Perry, K. O'Hara, A. Sellen, B. Brown, and R. Harper. Dealing with mobility: Understanding access anytime, anywhere. *ACM Transactions on Computer-Human Interaction*, 8(4):323–347, 2001.

G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

M. Pollack and M. Ringuette. Introducing the Tileworld: Experimentally Evaluating Agent Architectures. In T. Dietterich and W. Swartout, editors, *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 183–189, Menlo Park, CA, 1990. AAAI Press.

T. Rahwan, T. Rahwan, I. Rahwan, and R. Ashri. Towards A Mobile Intelligent Assistant: AgentSpeak(L) Agents on Mobile Devices. In P. Giorgini and M. Winikoff, editors, *Proceedings of the 5th International Bi-Conference Workshop on Agent Oriented Information Systems*, 2003.

A. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In W. V. de Velde and J. W. Perram, editors, *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *LNAI*. Springer, 1996.

A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI-Architecture. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, San Francisco, USA, 1995.

A. S. Rao and M. P. Georgeff. Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293–343, 1998.

R. Riggs, A. Taivalsaari, M. Vandenbrink, and J. Holliday. *Programming Wireless devices with Java 2 Platform Micro Edition.* Addison-Wesley Pub Co, Jun 2001.

P. S. Rosenbloom, A. Newell, and J. E. Laird, editors. *SOAR Papers: Research on Integrated Intelligence.* MIT Press, Cambridge, MA, USA, 1993.

N. Sadeh, T.-C. Chan, L. Van, O. Kwon, and K. Takizawa. Creating an open agent environment for context-aware m-commerce. In Burg, Dale, Finin, Nakashima, Padgham, Sierra, and Willmott, editors, *Agentcities: Challenges in Open Agent Environments*, LNAI, pages 152–158, Heidelberg, Germany, 2003. Springer Verlag.

N. Sahli. Survey: Agent-based middlewares for context awareness. *ECE-ASST*, 11, 2008.

S. Sardina, L. de Silva, and L. Padgham. Hierarchical Planning in BDI Agent Programming Languages: a Formal Approach. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1001–1008, New York, NY, USA, 2006. ACM.

M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–7. ACM Press, 1996. ISBN 0-89791-800-2.

B. N. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *Proc. of IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, California, USA, Dec 1994.

M. Schmill, D. Josyula, M. Anderson, S. Wilson, T. Oates, D. Perlis, D. Wright, and S. Fults. Ontologies for reasoning about failures in ai systems. In *Proceedings of the First International Workshop on Metareasoning in Agent-based Systems*, 2007.

N. Schurr, P. Patil, F. Pighin, and M. Tambe. Using multiagent teams to improve the training of incident commanders. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1490–1497, New York, NY, USA, 2006. ACM Press.

M. Schut, M. Wooldridge, and S. Parsons. The theory and practice of intention reconsideration. *Journal of Experimental & Theoretical Artificial Intelligence*, 16(4):261–293, October 2004.

T. B. Sheridan. Rumination on automation. In *Proceedings of 7th IFAC/IFIP/IFORS/IEA Symposium on Analysis, Design and Evaluation of Man-Machine Systems*, Kyoto, Japan, 1998. Oxford: Pergamon Press. Plenary address.

A. Singh and M. Conway. Survey of Context aware frameworks - Analysis and Criticism. Technical report, University of North Carolina, 2006.

R. Sun. *Cognition and Multi-Agent Interaction: From Cognitive Modeling to Social Simulation*, chapter The CLARION cognitive architecture: Extending cognitive modeling to social simulation, pages 79–102. Cambridge University Press, May 2008.

H. Takahashi, T. Suganuma, and N. Shiratori. AMUSE: An agent-based middleware for context-aware ubiquitous services. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems*, pages 743–749, Washington, DC, USA, 2005. IEEE Computer Society.

M. Tambe, E. Bowring, J. Pearce, P. Varakantham, P. Scerri, and D. Pynadath. Electric Elves: what went wrong and why. Technical report, American Association for Artificial Intelligence, June 2008.

M. B. van Riemsdijk. *Cognitive Agent Programming: A Semantic Approach*. PhD thesis, University of Utrecth, October 2006.

M. B. van Riemsdijk and J.-J. C. Meyer. A Compositional Semantics of Plan Revision in Intelligent Agents. In Springer-Verlag, editor, *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology (AMAST'06),*, volume 4019 of *LNCS*, pages 353–367, 2006.

G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1999.

M. J. Wooldridge. Intelligent agents. *Multiagent systems: a modern approach to distributed artificial intelligence*, pages 27–77, 1999.

M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, jun 1995.

H. Yan and T. Selker. Context-aware office assistant. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 276–279, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-134-8.

H. Zhang and S. Y. Huang. A general framework for parallel BDI agents. In *Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology*, pages 103–112, Washington, DC, USA, 2006. IEEE Computer Society.

Intelligente Mobiele Diensten met behulp van Agent Technologie

Zou het niet fijn zijn als je mobiele telefoon een intelligente persoonlijke assistent was? Eén die kon zien dat je in een vergadering zit en dus geen telefoon aan kan nemen. Die je documenten voor een vergadering voor je download als je van de ene naar de andere vergadering onderweg bent en dit stopt als blijkt dat je de vergadering gecanceled hebt. Of die onderweg naar huis vertelt dat er nog wat boodschappen gedaan moeten worden volgens een SMS van je partner en je vertelt van aanbiedingen van produkten op je lijstje van de supermarkt als je in de buurt bent van de winkel.

Om dit te bereiken creeerden we in dit onderzoek een computationeel model om intelligente mobiele services (IMS) te ontwikkelen. Intelligente mobiele services moeten aan drie voorwaarden voldoen. Ten eerste moeten ze zo autonoom mogelijk al het werk voor de gebruiker uitvoeren zodat er minimale interactie nodig is. Het systeem moet de goede informatie en service op de goede tijd aanbieden en zich aanpassen aan een snel veranderende omgeving. Ten slotte moeten IMS zuinig omgaan met de resources en zich bewust zijn van de beperkingen die mobiliteit met zich meebrengen (Batterij kan leeg raken, Internet verbinding valt weg, enz).

De architectuur voor de IMS is gebaseerd op agent technologie en in het bijzonder de BDI-agenten. Dit paradigma levert een goede basis voor dit werk omdat BDI agenten al autonoom kunnen werken en structuren hebben om te redeneren en kennis over de omgeving in op te slaan. Ze kunnen hun plannen en deliberaties echter niet snel aanpassen aan veranderende omgevingen (en blijven dus stug volharden in plannen die al achterhaald zijn). We hebben de architectuur van deze BDI agenten dus zodanig aangepast dat ze snel reageren als de gebruiker in een andere omgeving komt. Plannen kunnen worden afgebroken of opgeschort en nieuwe mogelijkheden snel worden benut. Tegelijkertijd worden de talloze signalen die worden opgevangen gefilterd zodat alleen die informatie die voor de gebruiker (op dat moment) van belang is wordt gebruikt voor het bepalen van de beste service.

Alhoewel er een prototype van de IMS op mobiele telefoons is geimplementeerd is er voor gekozen om de architectuur uitgebreid te testen in een simulatie omgeving waarin de reactie van het systeem op allerlei parameters uit de omgeving makkelijker kan worden bekeken. De resultaten zijn goed en de architectuur lijkt aan alle voorwaarden voor IMS te voldoen.

Curriculum Vitae

Fernando Luiz Koch

19 augustus 1971
Geboren te Curitiba-PR, Brazilië

mart 1989 - december 1993
Studie Informatica aan het Department of Information Systems van de Federal University of Santa Catarina, Florianopolis-SC, Brazilië. Diploma behaald in december 1993.

mart 1996 - october 1997
Master in de Informatica aan Departement van Informatie Systemen van de Federale Universiteit van Santa Catarina, Florianopolis-SC, Brazilië. Diploma behaald in october 1997.

mart 2002 - december 2008
Ph.D. studie aan het Instituut voor Informatica en Informatiekunde van de Universiteit Utrecht.

januari 2004 - mei 2005
Uitwisseling bij het Departement van Informatie Systemen van de Universiteit Melbourne, Melbourne, Australië.

SIKS Dissertatiereeks

1998

1998-1 **Johan van den Akker** (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 **Floris Wiesman** (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 **Ans Steuten** (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 **Dennis Breuker** (UM)
Memory versus Search in Games

1998-5 **E.W.Oskamp** (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 **Mark Sloof** (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 **Rob Potharst** (EUR)
Classification using decision trees and neural nets

1999-3 **Don Beal** (UM)
The Nature of Minimax Search

1999-4 **Jacques Penders** (UM)
The practical Art of Moving Physical Objects

1999-5 **Aldo de Moor** (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 **Niek J.E. Wijngaards** (VU)
Re-design of compositional systems

1999-7 **David Spelt** (UT)
Verification support for object database design

1999-8 **Jacques H.J. Lenting** (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation

2000

2000-1 **Frank Niessink** (VU)
Perspectives on Improving Software Maintenance

2000-2 **Koen Holtman** (TUE)
Prototyping of CMS Storage Management

2000-3 **Carolien M.T. Metselaar** (UVA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.
2000-4 **Geert de Haan** (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 **Ruud van der Pol** (UM)
Knowledge-based Query Formulation in Information Retrieval

2000-6 **Rogier van Eijk** (UU)
Programming Languages for Agent Communication

2000-7 **Niels Peek** (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 **Veerle Coup** (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 **Florian Waas** (CWI)
Principles of Probabilistic Query Optimization

2000-10 **Niels Nes** (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 **Jonas Karlsson** (CWI)
Scalable Distributed Data Structures for Database Management

2001

2001-1 **Silja Renooij** (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 **Koen Hindriks** (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 **Maarten van Someren** (UvA)
Learning as problem solving noindent 2001-4 **Evgueni Smirnov** (UM)

Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 **Jacco van Ossenbruggen** (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 **Martijn van Welie** (VU)
Task-based User Interface Design

2001-7 **Bastiaan Schonhage** (VU)
Diva: Architectural Perspectives on Information Visualization

2001-8 **Pascal van Eck** (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics.

2001-9 **Pieter Jan't Hoen** (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 **Maarten Sierhuis** (UvA)
Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design noindent 2001-11 Tom M.

van Engers (VUA)
Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01 **Nico Lassing** (VU)
Architecture-Level Modifiability Analysis

2002-02 **Roelof van Zwol** (UT)
Modelling and searching web-based document collections noindent 2002-03 **Henk**

Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 **Juan Roberto Castelo Valdueza** (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 **Radu Serban** (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 **Laurens Mommers** (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 **Peter Boncz** (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 **Jaap Gordijn** (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 **Willem-Jan van den Heuvel** (KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 **Brian Sheppard** (UM)
Towards Perfect Play of Scrabble

2002-11 **Wouter C.A. Wijngaards** (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 **Albrecht Schmidt** (Uva)
Processing XML in Database Systems

2002-13 **Hongjing Wu** (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 **Wieke de Vries** (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 **Rik Eshuis** (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 **Pieter van Langen** (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 **Stefan Manegold** (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 **Heiner Stuckenschmidt** (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 **Jan Broersen** (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 **Martijn Schuemie** (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 **Milan Petkovic** (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 **Jos Lehmann** (UVA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 **Boris van Schooten** (UT)
Development and specification of virtual environments

2003-07 **Machiel Jansen** (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 **Yongping Ran** (UM)
Repair Based Scheduling

2003-09 **Rens Kortmann** (UM)
The resolution of visually guided behaviour

2003-10 **Andreas Lincke** (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 **Simon Keizer** (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 **Roeland Ordelman** (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 **Jeroen Donkers** (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 **Stijn Hoppenbrouwers** (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 **Mathijs de Weerdt** (TUD)
Plan Merging in Multi-Agent Systems

2003-16 **Menzo Windhouwer** (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 **David Jansen** (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 **Levente Kocsis** (UM)
Learning Search Decisions

2004

2004-01 **Virginia Dignum** (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 **Lai Xu** (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 **Perry Groot** (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 **Chris van Aart** (UVA)
Organizational Principles for Multi-Agent Architectures

2004-05 `Viara Popova` (EUR)
Knowledge discovery and monotonicity

2004-06 **Bart-Jan Hommes** (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 **Elise Boltjes** (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 **Joop Verbeek** (UM)
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 **Martin Caminada** (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 **Suzanne Kabel** (UVA)
Knowledge-rich indexing of learning-objects

2004-11 **Michel Klein** (VU)
Change Management for Distributed Ontologies

2004-12 **The Duy Bui** (UT)
Creating emotions and facial expressions for embodied agents

2004-13 **Wojciech Jamroga** (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 **Paul Harrenstein** (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium.

2004-15 **Arno Knobbe** (UU)
Multi-Relational Data Mining

2004-16 **Federico Divina** (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 **Mark Winands** (UM)
Informed Search in Complex Games

2004-18 **Vania Bessa Machado** (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 **Thijs Westerveld** (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 **Madelon Evers** (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

2005

2005-01 **Floor Verdenius** (UVA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 **Erik van der Werf** (UM))
AI techniques for the game of Go

2005-03 **Franc Grootjen** (RUN)
A Pragmatic Approach to the Conceptualisation of Language

2005-04 **Nirvana Meratnia** (UT)
Towards Database Support for Moving Object data

2005-05 **Gabriel Infante-Lopez** (UVA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 **Pieter Spronck** (UM)
Adaptive Game AI

2005-07 **Flavius Frasincar** (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 **Richard Vdovjak** (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 **Jeen Broekstra** (VU)
Storage, Querying and Inferencing for Semantic Web Languages

2005-10 **Anders Bouwer** (UVA)
Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 **Elth Ogston** (VU)
Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 **Csaba Boer** (EUR)
Distributed Simulation in Industry

2005-13 **Fred Hamburg** (UL)
Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 **Borys Omelayenko** (VU)
Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 **Tibor Bosse** (VU)
Analysis of the Dynamics of Cognitive Processes

2005-16 **Joris Graaumans** (UU)
Usability of XML Query Languages

2005-17 **Boris Shishkov** (TUD)
Software Specification Based on Re-usable Business Components

2005-18 **Danielle Sent** (UU)
Test-selection strategies for probabilistic networks

2005-19 **Michel van Dartel** (UM)
Situated Representation

2005-20 **Cristina Coteanu** (UL)
Cyber Consumer Law, State of the Art and Perspectives

2005-21 **Wijnand Derks** (UT)
Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

2006-01 **Samuil Angelov** (TUE)
Foundations of B2B Electronic Contracting

2006-02 **Cristina Chisalita** (VU)
Contextual issues in the design and use of information technology in organizations

2006-03 **Noor Christoph** (UVA)
The role of metacognitive skills in learning to solve problems

2006-04 **Marta Sabou** (VU)
Building Web Service Ontologies

2006-05 **Cees Pierik** (UU)
Validation Techniques for Object-Oriented Proof Outlines

2006-06 **Ziv Baida** (VU)
Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 **Marko Smiljanic** (UT)
XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 **Eelco Herder** (UT)
Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 **Mohamed Wahdan** (UM)
Automatic Formulation of the Auditor's Opinion

2006-10 **Ronny Siebes** (VU)
Semantic Routing in Peer-to-Peer Systems

2006-11 **Joeri van Ruth** (UT)
Flattening Queries over Nested Data Types

2006-12 **Bert Bongers** (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 **Henk-Jan Lebbink** (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 **Johan Hoorn** (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 **Rainer Malik** (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 **Carsten Riggelsen** (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 **Stacey Nagata** (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 **Valentin Zhizhkun** (UVA)
Graph transformation for Natural Language Processing

2006-19 **Birna van Riemsdijk** (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 **Marina Velikova** (UvT)
Monotone models for prediction in data mining

2006-21 **Bas van Gils** (RUN)
Aptness on the Web

2006-22 **Paul de Vrieze** (RUN)
Fundaments of Adaptive Personalisation

2006-23 **Ion Juvina** (UU)
Development of Cognitive Model for Navigating on the Web

2006-24 **Laura Hollink** (VU)
Semantic Annotation for Retrieval of Visual Resources

2006-25 **Madalina Drugan** (UU)
Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 **Vojkan Mihajlovic** (UT)
Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 **Stefano Bocconi** (CWI)
Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 **Borkur Sigurbjornsson** (UVA)
Focused Information Access using XML Element Retrieval

2007

2007-01 **Kees Leune** (UvT)
Access Control and Service-Oriented Architectures

2007-02 **Wouter Teepe** (RUG)
Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 **Peter Mika** (VU)
Social Networks and the Semantic Web

2007-04 **Jurriaan van Diggelen** (UU)
Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 **Bart Schermer** (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 **Gilad Mishne** (UVA)
Applied Text Analytics for Blogs

2007-07 **Natasa Jovanovic** (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 **Mark Hoogendoorn** (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 **David Mobach** (VU)
Agent-Based Mediated Service Negotiation

2007-10 **Huib Aldewereld** (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 **Natalia Stash** (TUE)
Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 **Marcel van Gerven** (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 **Rutger Rienks** (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 **Niek Bergboer** (UM)
Context-Based Image Analysis

2007-15 **Joyca Lacroix** (UM)
NIM: a Situated Computational Memory Model

2007-16 **Davide Grossi** (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 **Theodore Charitos** (UU)
Reasoning with Dynamic Networks in Practice

2007-18 **Bart Orriens** (UvT)
On the development an management of adaptive business collaborations

2007-19 **David Levy** (UM)
Intimate relationships with artificial partners

2007-20 **Slinger Jansen** (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 **Karianne Vermaas** (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 **Zlatko Zlatev** (UT)
Goal-oriented design of value and process models from patterns

2007-23 **Peter Barna** (TUE)
Specification of Application Logic in Web Information Systems

2007-24 **Georgina Ramrez Camps** (CWI)
Structural Features in XML Retrieval

2007-25 **Joost Schalken** (VU)
Empirical Investigations in Software Process Improvement

2008

2008-01 **Katalin Boer-Sorbn** (EUR)
Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 **Alexei Sharpanskykh** (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 **Vera Hollink** (UVA)
Optimizing hierarchical menus: a usage-based approach

2008-04 **Ander de Keijzer** (UT)
Management of Uncertain Data - towards unattended integration

2008-05 **Bela Mutschler** (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 **Arjen Hommersom** (RUN)
On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 **Peter van Rosmalen** (OU)
Supporting the tutor in the design and support of adaptive e-learning

2008-08 **Janneke Bolt** (UU)
Bayesian Networks: Aspects of Approximate Inference

2008-09 **Christof van Nimwegen** (UU)
The paradox of the guided user: assistance can be counter-effective

2008-10 **Wauter Bosma** (UT)
Discourse oriented summarization

2008-11 **Vera Kartseva** (VU)
Designing Controls for Network Organizations: A Value-Based Approach

2008-12 **Jozsef Farkas** (RUN)
A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 **Caterina Carraciolo** (UVA)
Topic Driven Access to Scientific Handbooks

2008-14 **Arthur van Bunningen** (UT)
Context-Aware Querying; Better Answers with Less Effort

2008-15 **Martijn van Otterlo** (UT)
The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.

2008-16 **Henriette van Vugt** (VU)
Embodied agents from a user's perspective

2008-17 **Martin Op 't Land** (TUD)
Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 **Guido de Croon** (UM)
Adaptive Active Vision

2008-19 **Henning Rode** (UT)
From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 **Rex Arendsen** (UVA)
Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie

van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven

2008-21 **Krisztian Balog** (UVA)
People Search in the Enterprise

2008-22 **Henk Koning** (UU)
Communication of IT-Architecture

2008-23 **Stefan Visscher** (UU)
Bayesian network models for the management of ventilator-associated pneumonia

2008-24 **Zharko Aleksovski** (VU)
Using background knowledge in ontology matching

2008-25 **Geert Jonker** (UU)
Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 **Marijn Huijbregts** (UT)
Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 **Hubert Vogten** (OU)
Design and Implementation Strategies for IMS Learning Design

2008-28 **Ildiko Flesch** (RUN)
On the Use of Independence Relations in Bayesian Networks

2008-29 **Dennis Reidsma** (UT)
Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 **Wouter van Atteveldt** (VU)
Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 **Loes Braun** (UM)
Pro-Active Medical Information Retrieval

2008-32 **Trung H. Bui** (UT)
Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 **Frank Terpstra** (UVA)
Scientific Workflow Design; theoretical and practical issues

2008-34 **Jeroen de Knijf** (UU)
Studies in Frequent Tree Mining

2008-35 **Ben Torben Nielsen** (UvT)
Dendritic morphologies: function shapes structure

2009

2009-01 **Rasa Jurgelenaite** (RUN)
Symmetric Causal Independence Models

2009-02 **Willem Robert van Hage** (VU)
Evaluating Ontology-Alignment Techniques

2009-03 **Hans Stol** (UvT)
A Framework for Evidence-based Policy Making Using IT

2009-04 **Josephine Nabukenya** (RUN)
Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 **Sietse Overbeek** (RUN)
Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 **Muhammad Subianto** (UU)
Understanding Classification

2009-07 **Ronald Poppe** (UT)
Discriminative Vision-Based Recovery and Recognition of Human Motion

2009-08 **Volker Nannen** (VU)
Evolutionary Agent-Based Policy Analysis in Dynamic Environments

2009-09 **Benjamin Kanagwa** (RUN)
Design, Discovery and Construction of Service-oriented Systems

2009-10 **Jan Wielemaker** (UVA)
Logic programming for knowledge-intensive interactive applications

2009-11 **Alexander Boer** (UVA)
Legal Theory, Sources of Law & the Semantic Web

2009-12 **Peter Massuthe** (TUE, Humboldt-Universitaet zu Berlin)
Operating Guidelines for Services

2009-13 **Steven de Jong** (UM)
Fairness in Multi-Agent Systems

2009-14 **Maksym Korotkiy** (VU)
From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)

2009-15 **Rinke Hoekstra** (UVA)
Ontology Representation - Design Patterns and Ontologies that Make Sense

2009-16 **Fritz Reul** (UvT)
New Architectures in Computer Chess