Master of Science

Supporting impromptu coordination
using intelligent software and mobile devices

Utrecht University

Author:
Andrew Koster

Thesis no:
INF/SCR-06-18

Supervisors:
Frank Dignum
John-Jules Meyer

Date of Submission:
April 4, 2007

Abstract

One of the challenges in contemporary AI research is dealing with highly dynamic environments. In this thesis we consider the problem of supporting impromptu coordination in just such a dynamic environment: the daily use of a mobile device. We outline the various contexts with which we deal daily and how they change. We show how a context-aware agent can be of use in helping with this. We create a logical framework, in which we describe the problem of impromptu coordination and an agent system to deal with it. We also describe a Multi Agent System (MAS) framework, with three different kinds of agents: context-agents, a brain-agent and an intention-agent, which together are capable of dealing with the problem of impromptu coordination. We explain how the deliberation cycle works and outline the most important algorithms used.

The main benefit of the MAS architecture we describe is the fact that deliberation is distributed over the agents. Each agent deals with its own encapsulated part of the greater problem and not all of these agents have to run on the mobile device. We outline the benefits of this approach, as well as the pitfalls, and then show how our prototype implementation of this MAS can deal with some examples.

Contents

Preface

After taking much too long I am very glad to say this thesis is finally finished. It took a lot of work, as well as a lot of help. Firstly I'd like to thank Frank Dignum for enabling my stay in Australia and Liz Sonenberg and the Department of Information Systems of the University of Melbourne for letting me do my research there. I am very grateful to Iyad Rahwan, Liz Sonenberg, Samin Karim and Fernando Koch for their inspiration and guidance during the early stages of this project.

I am greatly indebted to Frank Dignum and John-Jules Meyer for their supervision when I was back in Utrecht. Their ideas and comments made me examine my own research critically and sharpened my wits.

Finally a word of thanks to my family and friends for their support and listening ear when I explained my problems and ideas time and time again. Thanks to Mathijs Booden's proofreading, the use of language in my thesis is that much better. I especially respect him for plowing through the entire work without knowing the first thing about Artificial Intelligence, or even Computer Science in general. Especial thanks to my fellow students Stefan Leijnen and Silvain van Weers for our many interesting discussions and research projects.

Chapter 1

Introduction

Physicists have determined that even the most solid and heavy mass of matter we see is mostly empty space. But at the submicroscopic level, specks of matter scattered through a vast emptiness have such incredible density and weight, and are linked to one another by such powerful forces, that together they produce all the properties of concrete, cast iron and solid rock. In much the same way, specks of knowledge are scattered through a vast emptiness of ignorance, and everything depends upon how solid the individual specks of knowledge are, and on how powerfully linked and coordinated they are with one another.

—T. Sowell

In the past few years there has been a continuing trend towards the use of mobile (wearable) computers in everyday life. Because of the development of more advanced wireless communication systems, these mobile computers are becoming more and more *connected*. Connected to other computers, but also to devices that provide real-time environmental information, such as GPS data. We will focus on how all this information can be incorporated into a mobile, connected, situation aware application to support impromptu coordination in everyday life. Our main focus is not the coordination of multiple users, but rather the enabling technology to help coordinate the daily schedule of a single user. More specifically, we wish to support the ability for on-the-fly decision making, rather than long term planning, for which we believe there are already sufficient other techniques developed and in development.

1.1 On impromptu coordination

What do we mean with impromptu coordination?

The Webster dictionary says the following about **coordination**:

> The act of regulating and combining so as to produce harmonious results; harmonious adjustment.

To obtain these harmonious results we need to communicate with other people, have background information and interact with the dynamic world around us. We schedule meetings, coordinate lunch appointments with friends, have dinner in time to watch our favourite TV shows and meet our friends at the local pub. Most of this can be planned or is part of the daily routine, but situations change and during the day new opportunities arise. These are opportunities for ***impromptu** coordination*, with which we mean coordination resulting from unprepared (and possibly unwanted) changes in the *context* we use to coordinate our busy lives.

Dealing with impromptu coordination is therefore far more complex than dealing with coordination or planning in general. Opportunities for coordination arise in a highly dynamic world and must be dealt with on-the-fly. There is generally little time to consider many alternatives and ad-hoc decisions need to be made, optimizing your gains without much calculation of alternative plans. A lot of the situations that arise will require a coordinated schedule with other people (meetings, car pools, social events). However, this is not the part we choose to focus on. This kind of coordination requires sophisticated negotiation techniques, which is not what we wish to research. We wish to focus on the single-user situation, where we disregard the ability to change other people's plans, while instead accepting other peoples' plans and schedules as a part of the environment and available information. A context which may change, but which we take no active part in changing. An inspiring approach to the problem of coordinating in multi-user systems is interest-based negotiation, discussed in Iyad Rahwan's dissertation [27]. Our initial approach in this research was based on his framework, but instead of using interest-based negotiation, we aimed to enable multi-user impromptu coordination using traditional negotiation methods such as contract net[34]. However, the single-user problem precludes this approach already. To be able to negotiate it is imperative to know what you wish to negotiate about. In a highly dynamic environment, such as that encountered in mobile devices, goals and environment are in constant change. The problem of single-user coordination is therefore complex enough without considering the added ramifications of a multi-user system. So we pushed that problem to the background and treat it as the environment (or more specifically, as one of the contexts we must be aware of, see 1.1.1), however we believe the framework discussed here can be naturally extended with negotiation techniques to tackle the multi-user problem.

1.1.1 Context awareness

Before we can move on to the describing of a framework, we must define the problem space more clearly. There are many different opinions on what context and context-awareness entails. We have chosen to use the definition of context given by Chen and Kotz in [5]:

> Context is the set of environmental states and settings that either determines an application's behavior or in which an application event occurs and is interesting to the user.

The type of context we are interested in is the first type, called *active context*. An application is *active context-aware* if:

> An application automatically adapts to discovered context, by changing the application's behavior.

Because contextual awareness in general is a very difficult problem[21] we will focus only on the situations we feel are valuable in day-to-day use of a mobile device. More specifically, the situations we will focus on are:

Time One of the main uses of mobile devices is its calendar function. We can use this information so our application may be aware of the user's appointments and deadlines. The computer's system clock gives us information about the current time, which can be used in planning.

Location Given a map and locations a smart program can plan a route between them, calculate the time needed to travel this distance and find other locations near these spots that may be of interest. It may even use different means of transport available to the user to find optimal results [16].

Friends and colleagues Your mobile device will contain an address book and/or instant messenger program. The information found in this can be grouped and used. Some people will be grouped as friends, some as colleagues, strangers, family, etc. Their contact information is attached, thus they can be contacted as well.

Plans and desires We need the application to be aware of its own plans and desires, to be able to adapt the plans and adapt to changes in desires. We need our agent to be introspective.

Other agents Connected mobile devices is a domain in which a lot of actions depend on other agents (users and/or computers acting on their behalf). We will need to be aware of these other agents and whatever information they care to share with us.

Online information More and more programs and webpages are becoming aware of the use of automated search agents. This information is supplied within a shared ontology, with hierarchical information. This can be used by the agent to 'understand' what services are being provided[21].

Personal information To make our agent useful to us we will need to customize it. We can use an ontology to describe the desires of the user and give background information about him/her. Being aware of such information lets the agent be more attuned to the user.

These are the different types of context we will distinguish. There may be more, but we discern these as the more important ones for impromptu coordination in daily situations. We use these, because we feel that these encompass a wide scala of uses for mobile devices. Changes in schedule, moving around, information about people around you and changes in your own plans and desires make up most of the changes that create opportunities for impromptu coordination, while the internet and any personal information available provide a background which can be used to evaluate ways to take advantage of those opportunities.

1.2 Changes in context

Now that we have a definition of context and what types of context we wish to be aware of, we can focus more specifically on how these contexts may change. More specifically what it means when the context changes for a situated aware application. Basically we discern four different levels at which context changes influence our environment in such a way that they change the way we should or can act, thus giving rise to a possibility for impromptu coordination. We base our actions on what we want (desires) and information/knowledge about the world around us. With this knowledge we form plans, which require 'resources' to fulfill.

Each of the above types of context can change, resulting directly in one of the following changes for a rational agent:

- Change in available *information/knowledge*
- Change in *resources*
- Change in *plans*
- Change in *desires*

These changes are exactly what we identify as giving rise to impromptu coordination. Changes in desires are generally solved using traditional planning techniques, so we won't discuss them further as one of the changes in context. In practice an agent must of course *also* be able to plan in the eventuality of new desires arising. The framework we design is able to do so, however for now we will focus on how the other three types of change impact an agent.

1.2.1 Resources

With resources we mean all material means necessary to accomplish a task. We also include time as a resource to make the discussion easier and clearer. To understand what we mean by changing resources, consider the following example:

> Your car breaks down on your way to work. Your goal to get to work is a high priority, so you need to find alternate resources, such as going by bus.

1.2.2 Information

You may have to change your plans, because you learn something about the world around you, that enables plans to fulfill previously unobtainable desires.

For example: we learn that the Melbourne Music Festival is opening at 6 PM tonight; we have no specific tasks scheduled for the night and can therefore go there, fulfilling the desire of 'sniffing culture'.

We do not wish to categorize this as a new resource being found (namely "The existence of the Melbourne Music Festival"), because the actual handling of new information is significantly different. Acquiring new information must go through a processing step, in which we discover whether it is *relevant* or not. If it is, it will map out to a change in resources.

Other information may open up resources that have been available all along, just out of reach. An example of this is location information on maps. For example, if you want an ice cream and know of some ice cream stalls around town, you might still not go to them, because they're too far away. If, on the other hand, you have to go to a meeting and come near one of them, that resource becomes available by making only a slight detour. This is because your location changed, or in a more abstract sense, the *information* about your location changed.

1.2.3 Plans

An agent's own plans may change. He may add new plans that are better than old ones, or simply change existing plans. This may free up resources, which enables the agent to accommodate for other plans. Thus changes to an agent's own plans can be perceived as a slightly more complicated form of resource changes, where some resources may get used up, while others are freed for use. Changes in other agents' plans, have to be treated completely separately. Other people's plans (of which your plans are dependent) may change as well and you may have to accommodate for these, or they create opportunities for you to cooperate, by affecting shared resources. An example of changing plans is: your boss reschedules a meeting, leaving you time to go out for a lunch appointment.

1.2.4 Conclusion on contextual changes

The previous list of context changes in 1.2 may seem fairly straightforward; in practice changes in the contexts discerned in 1.1.1 will often map to a combination of the changes in the agent. A change in schedule may impact both the available time (resources) as well as be a change of plans. These different changes will ultimately map to changes in resources, which may even overlap. We will see in chapter 2 how this works. The reason we treat them separately is because intuitively we see them as different cases. Getting information about the Melbourne Arts Festival and having a meeting rescheduled are conceptually different, even if they both may be traced to changing resources, that influence your achievable desires.

1.3 Why mobile devices?

While you may sometimes encounter opportunities for impromptu coordination in a static environment, it is much more likely that you will be on the move when such opportunities arise. The advantage of running on a mobile device is that it usually travels with you. It also offers connectivity to the internet and telephone networks, thus giving you access to information you may need to help you coordinate. A mobile device also gives you the means to communicate with other people while on the move. The newest technologies give human beings an unprecedented opportunity to access their offices, homes and the vast information storage on the internet from almost anywhere. The challenge lies in how to create software that will support them.

Not only is the connectivity a valuable asset, but most mobile devices retain a wealth of information themselves. Calendar functionality and route planners are obvious examples of contextual information available on the device.

1.4 Characteristics of impromptu coordination

The problem of supporting impromptu coordination with mobile devices has been researched in many situations already [28, 29], most notably in the ActiveCampus environment [12]. From these studies certain characteristics of impromptu coordination emerge:

Dynamic The environment in which the mobile user is working is highly dynamic. This means that any solution must be able to deal with this. In [17] it is argued that modern societies show a high amount of *spatial*, *temporal* and *contextual* mobility. While a lot of research is already being done into both spatial and temporal mobility, we will focus on the problem of contextual mobility. Wherever we go, the context on which we base our decisions changes. This creates new opportunities for coordination.

Privacy and Connectivity Another important issue in mobile societies is the fact that more and more information may become available to other users. Where in more primitive societies privacy was easily controlled, in the modern connective world it is becoming a larger issue. Any method for solving impromptu coordination must take this into account and the user must be able to choose how much information to publicize.

Heterogeneity Everybody solves their problems in their own manner, so everyone may achieve his/her desires in a different way. Because of the incredibly diverse amount of information and different resources available there are often many ways of achieving the same goal. Because of the large scale of different desires and means to fulfill them, there is a large possibility of conflicts arising. These must be resolved to achieve the greater goal of taking advantage of the opportunities offered by the available resources.

1.5 Why agents?

After discerning the various different situations in which the need for impromptu coordination arises and seeing that mobile devices may be of great use in these, we ask ourselves whether 'agents' can help us. We will use the definition of an agent as given by Woolridge and Jennings [38]:

> An ***agent*** is a computer system that is *situated* in some *environment* and that is capable of *autonomous action* in this environment in order to meet its *design objectives*.

We wish to design an application that can assist a person with his impromptu decision making. To do this we need some rudimentary model of his wishes and decision making process. Agents are ideally suited for this, being goal oriented autonomous programs. While active databases[26] could possibly approach the same context-awareness, the real power is in the utilization to forward one's goals. This is the strength of an agent-oriented approach: the ability to use the context-awareness to fulfill one's desires.

To return to Wooldridge and Jennings' definition, in our approach the *environment* would be as much of the 'real world' as the agent can access through the mobile device, typically a calendar for scheduling, a messaging program, an address book and a map. It should also be able to access a certain amount of information and shops on the internet. The problem is what to make of all this information; the agent has to be somehow aware of the contextual value of all of this. A calendar will say something about the user's schedule and a map may have his position. Based on all this contextual information, the agent's *autonomous actions* would be to help you with on-the-fly decision making. It could advise courses of action or undertake them itself if these were its *design objectives*.

This is an obvious evolution from the current mobile device, which passes all the information on to the user. An agent may first process this information autonomously and only pass on the useful information/advice to the user. Some examples of the use of the functionality of such an agent are:

1. You have a PDA with an intelligent personal agent. It has your calendar and therefore knows you have the evening free. It also has a map and knows you're in Melbourne. It has enough knowledge about you to know you're interested in cultural events. It searches the web and finds out that it's the Melbourne festival's opening night. It flashes active and suggests you might go there. It contacts the PDAs of friends whose interests are similar and gives you a list of people who might be interested in accompanying you.

2. Your car breaks down, sending an automatic message to your PDA notifying it. Your intelligent agent checks where you're going in the calendar and accesses the internet to find the nearest bus stop with a bus going to work. It suggests a route to walk and tells you when the bus is leaving.

3. You have your agent schedule a lunch date with friends, but this fails, because a planned meeting with your boss is in the way. Your boss' agent contacts yours during the day to reschedule the meeting; leaving space for lunch, which your PDA replans, taking time constraints and restaurant preferences into account.

1.5.1 BDI Agents in dynamic environments

Some research has been done into the development of agent systems for mobile devices [30, 22], yet it is still very much an emerging field of research. BDI-Architectures seem particularly suited to this domain, due to their ability to cope autonomously in dynamic situations. However, as O'Grady and O'Hare point out in [22] there are some fundamental differences between developing agents for use on mobile devices and for desktop systems. The computational resources available are much more limited, while the environment in which the agent must act is far more dynamic. Analogue to our research O'Grady and O'Hare developed a Multi-Agent System for dealing with such a dynamic environment, particularly for use as a mobile tourist guide. Their application of a multi-agent system is similar to that of the GUIDE project[6], in that they wish to make an interactive location-aware tourist guide. However, in contrast to GUIDE they focus on an agent-oriented approach to dealing with the complex and dynamic environment. They do, however, only take the location of the user into account and a lot of the complexity we deal with every day is in combining all the information at our disposal. This is not necessary for a tourist guide, but in our approach it is an integral part of the problem. We aim at the day to day use of a PDA as a useful 'companion' when sorting out the complex coordination arising in a dynamic environment, specifically the complexity of combining information from different contexts.

1.5.2 Agent solutions for impromptu coordination

In the previous section we took note of the specific characteristics of impromptu coordination. We think agents are especially suited to deal with these characteristics. Agents can support decision making successfully already [4] and are a useful abstraction of the way humans think. Dynamic situations require an internal representation of the world and an awareness of the contextual environment the program is placed in. Context-aware situated agents give a natural architecture for dealing with this. The hierarchical way in which they may order information also gives a solution for problems arising from heterogeneity. The conflicts must be solved, based on relevance, authority and importance, all of which can be incorporated in the agents' beliefs.

As a personal agent is in its deepest sense an abstraction of the user, it may also as easily be programmed with any privacy requirements the user has, while exploiting the information provided by connectivity to discover opportunities.

1.6 A precise formulation of the problem

An agent running on a PDA will be dealing with a variety of specific contexts and it will be programmed with specific desires (that the user may change at any time), which it should try to fulfill. There may be different types of desires, with different priorities (going to a meeting with your boss may be more important than having lunch with a friend), but each one is a desirable outcome for the user and we want to achieve them. We now define the problem of impromptu coordination as:

> **How can an agent recognize and help react to opportunities for fulfilling desires in changing contexts?**

The main goal of the agent is to assist the user in such situations where opportunities arise. In doing this the agent's main output will be advice to the user. Depending on how much autonomy the user chooses to give the agent it may act more or less autonomously, doing such things as rescheduling appointments in the calendar, contacting similar agents' acting for friends or partners, requesting web services and using other *digital resources* to fulfill the user's desires as efficiently as possible.

1.7 Human-Computer Interaction

An important issue in the question of whether Mobile Agents are useful for solving impromptu coordination, is the one studied by Human-Computer Interaction-researchers. What and how do users want the program to solve tasks? The general view is that a program should support users in how they do things, but not take over control of what they do [23]. Graham and Cheverst [11] discern 5 basic roles that a mobile guide may take to support its user. The 'buddy role' is described as:

> A *Buddy* has elements of the expert system, decision support system and information repository. The Buddy utilizes mixed initiative dialogues, sharing control of the interaction with users. A Buddy exhibits a high level of 'intelligence' concerning its interaction with the user

This is a role we can imagine a supporting agent taking. It should be able to decide mundane things itself, but when needing user input, ask intelligent questions. The autonomy required of an agent may also depend on the specific user's preferences. Some users may require total control, in which case the agent is no more than an intelligent source of information, while others may give the agent free reign to automatically take advantage of any opportunity that may arise. We presume that most users will be found in the middle, using the agent, but not giving it complete control. Future research must show whether this assumption is true and to what degree it is dependent on how the agent is developed. We will therefore need to provide a certain amount of settings, corresponding to the scaling autonomy of the agent. More on this is to be found in [2] and chapter 3 deals with it briefly in our framework.

Our main research area isn't HCI, however and we don't presume to be doing any new work in this area. We just wish to point out that there is relevant research being done relating to our project and in an implementation it should be taken into account. The buddy paradigm seems ideally suited to the way we intend a supporting agent to interact with its user.

The same goes for our claims about scalability of autonomy. We won't do research into how scalable autonomy fits into the supporting agent, but instead just show that using different plans that each require more or less control from the user a preprogrammed scalable autonomy can be accomplished. The appropriate plans are selected according to the settings the user selects, not through any learning from the user. For a more complete view on using scalable autonomy in agents, see [4].

1.8 Example applications

Now that we have given an introduction, we will discuss some complex examples of impromptu coordination in a dynamic environment. We wish to do this before moving on to the more technical aspect of the problem, as well as the solution, so as to give a more down-to-earth insight into the challenge we face.

1.8.1 A lunch date

We have already touched on this example a couple of times, but not yet in its full complexity. It is essentially a multi-user problem, but we will treat the plans and desires of other users as contextual information.

You have a group of friends: John, Sarah and Peter, who you haven't seen in a while and wish to meet up with soon. However, all of you are busy this week and they decide to meet for lunch on a day when you have an important meeting with your boss, unfortunately at lunch time. On the fateful day your boss phones at 11:30 AM to cancel the meeting. You phone John to let him know you can meet them for lunch after all, but in the meantime Peter has gotten sick, so Sarah and John had decided to cancel lunch as well. John instead scheduled a dentist appointment at 1:30 PM and Sarah has to drop her daughter off at school at noon. You manage to coordinate an appointment for 12:15 at a nice bistro near Sarah's daughter's school, when Sarah calls to remind you she is a vegetarian. You relocate to a deli near John's dentist at 12:45 and manage to meet for lunch.

This example is interesting, because it is a situation that most people can recognize as costing a lot of time, while appearing extremely simple. There are a large amount of phone calls being made only to organize a simple lunch date, as well as the research work needed to find the place for the appointment. There are some strict conditions and there is little time to organize it. Firstly there is the difference between relevant and irrelevant information. At first the information that John, Sarah and Peter had canceled their appointment was irrelevant, because you couldn't be there anyway. However, once the meeting with your boss had been canceled, it suddenly was relevant, as were John and Sarah's new plans. These new plans put some strict conditions on the appointment: it had to be between 12 and 1:30, leaving sufficient time to travel. Thus if it was early after 12 it had to be near the school, or if it was later it needed to be near the dentist. In addition there is the background information about Sarah's stricter diet, which excluded the bistro as a viable place to have lunch. This combines 6 of the contexts discussed in 1.1.1: time, location, friends' and colleagues' plans, your own plans, information about other agents (Sarah being a vegetarian) and background information (location and menu of different restaurants). A lot of the information used is normally not easily available to a user, however it is fairly easy to access for a computer system with sufficient resources. In this way the scheduling problems could have been simplified or even avoided entirely.

1.8.2 Buying a book

This is a simpler example than the one above, but emphasizes the importance of situated awareness in a dynamic environment, rather than the complexity of combining different contexts.

You wish to buy a book and know of various different book stores. First you phone the local bookstore, who are redoing their inventory system and don't know whether the book is in stock, but can give you a price. You have an appointment in the afternoon on the wrong side of town, so you check nile.com and stable-sandbase.com, which are online bookstores. You find they are cheaper than the local bookstore and have a delivery time of 5 days. You would prefer to have it sooner than that, but the express delivery rate is significantly more expensive. Before placing the actual order, your wife calls to say she has an important meeting and needs you to pick up the children from school. This takes you near the bookstore, who did indeed receive the book in the afternoon shipment and you can pick it up.

In this example the problem was not so much to do something that was previously impossible, but rather to make the best choice given the available information. At first the uncertain availability as well as the distance to the book store were prohibitive to buying the book. However, the change in schedule that forced you to go near the book store, enabled you to check into the shop to simply check whether they had it, getting it immediately at a lower price than the express delivery from the online stores.

This shows a change in priorities rather than in possible plans. However, if you had already ordered the book from an online store, this would not have been relevant, due to your commitment to the current approach. Only if the online store could then not deliver would you once again consider other options.

1.9 This document

This document is divided into 5 chapters. You have just read the first chapter. In chapter 2 we will take a closer look at the problem and give a formal description of it. We will also give a formal logic for an agent system dealing with impromptu coordination, while in chapter 3 we will describe an architecture for such a system. In chapter 4 we will demonstrate the workings of a prototype implementation of such a system, concluding the thesis with our results and possible future work on this subject in chapter 5.

Chapter 2

A precise formulation of impromptu coordination

In the previous chapter we discussed the problems involved with impromptu coordination. Here we will give a more formal description from a computer scientific point of view. We will use a PDA as an example device, but it could be any connected mobile device.

2.1 A formal agent

For a formal definition of impromptu coordination, we will need an agent to have certain attributes. Most are commonly used in agent-based systems and we will only explain those of specific interest to us. We will loosely base our agent theory on Wagner's Vivid Agents theory [35] and the BDI Agent theory[32, 37]. The logics used for resources are similar to those used by Mathijs de Weerdt[9].

Let $\mathcal{L}$ be a first order logical language with $\wedge$ for conjunction, $\neg$ for negation and $\exists$ as the existential quantifier. We will also use the usual abbreviations $\vee, \rightarrow, \perp, \forall$ as well as $\vdash$ for classical and $\models$ for semantic inference.

2.1.1 Beliefs

We break with classical BDI theory by making no distinction between beliefs and knowledge, assuming that an agent is always subjective and (like humans) cannot make the distinction between belief and *true* belief (knowledge). We therefore approach the idea of beliefs based on the domain we're trying to model: the beliefbase is what an agent believes/knows about the world around it. Beliefs are abstract representations of the context in which the agent is situated. For each different kind of context a different representation may be adequate. While an address book is best modeled as some kind of database, a location may be better modeled in some kind of GIS structure.

In addition, each belief is associated with a probability p of being a true statement about the real world. This is useful when adding new beliefs that may conflict, on which more in 2.2.1.

Our beliefbase $\mathcal{B}_A$ is the union of all these different representations with the following requirements.

Inner consistency rule Each representation type for beliefs B that is used in the beliefbase must have a formal definition of consistency. $\mathcal{L}$ may therefore be used, as well as relational databases, calendars and any other datastructure that has a rule for consistency.

Outer consistency rule For each set of datastructures $(B_1, B_2, \ldots, B_n)$ with $\forall i \leq n : B_i \in \mathcal{B}_A$ there must be a non-trivial boolean relation $\vdash_{B_1 \ldots B_n}$ such that $\bigcup_{i \leq n} B_i \nvdash_{B_1 \ldots B_n} \perp$.

Together these rules create the global consistency of our beliefbase. If an agent has belief $\phi \in R$, the inner consistency rule assures that $\forall \psi \in R : \phi \wedge \psi \not\models \perp$, while the outer consistency rule assures $\forall R' \in \mathcal{B}_A : \forall \psi \in R' : \phi \wedge \psi \not\models \perp$. Most specifically for atomic beliefs: if $\phi \in \mathcal{B}_A$ then $\neg\phi \notin \mathcal{B}_A$. At first sight, this looks like an extremely complicated way of doing things. In particular, note that the *outer consistency* is not a well-defined rule, but from a design perspective this makes sense. Outer consistency will not usually be implemented as a formal relation, but rather as a complete disjunction of functionality of two different data types. We can say nothing about consistency when comparing apples and oranges, so how can we when comparing location and a bus timetable? We only include the rule for outer consistency for those cases in which the data *is* comparable. We can, for instance, require that if two different datastructures contain location information and they both contain the location for the same object, then that location is the same too. Furthermore some datastructures may not use the assumption of a closed world, but use an open world structure. In this case we have a representation for sentences known to be true and another similar representation for sentences known to be false. If a sentence is in the first set, it may logically not be in the second.

Another reason to represent beliefs in this manner is that doing it differently would require some double representations of data on the PDA. A PDA usually has a perfectly good calendar and address book on it already, while memory is at a premium. We therefore need to design an agent that makes efficient use of the available structures.

2.1.2 Desires

Desires are handled conventionally, but with an extension of what kind of desire they are. Thus an agent's desires are a set
$\mathcal{D}_A = \{(\delta_1, desc_1, worth_1), (\delta_2, desc_2, worth_2), \ldots, (\delta_n, desc_n, worth_n)\}$, where $\bigcup_{i \leq n} \{d_i\} \subseteq \mathcal{D}$, with $\mathcal{D}$ the set of all possible desires as atomic propositions in

$\mathcal{L}$. $worth_i \in [0,1]$ is the worth of fulfilling the desire δ_i and $desc_i$ is one of the following:

- one-off
- recurring(n), with $n \in \mathbb{N}$ a time.
- continuous

We will see how this is used later on.

2.1.3 Resources

Resources are physical properties of the environment that are available to the agent.

Definition 2.1 *A resource tuple is a tuple $R := \langle \rho, cost \rangle$, where $\rho \in \mathcal{R}$ with $\mathcal{R}$ all resources available in the world and $cost \in [0,1]$ the abstract cost of using the resource (may differ per agent). A resource $\rho \in \mathcal{R}$ is an atomic proposition, possibly with parameters, for example* $\texttt{free\_time}(\texttt{me}, \texttt{now} - 4 : \texttt{30pm})$ *or* $\texttt{distance}(\texttt{Berry\&Jens}, 35)$
An agent's resourcebase is now defined as: $\mathcal{R}_A = \{R_1, R_2, \ldots, R_m\}$ with $\forall i \in \{1, \ldots, m\} : R_i$ a resource tuple.

We use $cost(\rho), \mathcal{A}$ to denote the cost of using resource ρ to agent $\mathcal{A}$, and $available(\mathcal{A}) = \{\rho \in \mathcal{R} | \langle \rho, cost \rangle \in \mathcal{R}_A\}$ to denote all resources available to $\mathcal{A}$. This resourcebase will be constructed through an inductive process shown later. The truth value of a resource ρ will signify its availability. If ρ is true it means the resource is known to be available, while if $\neg\rho$, it means the resource is known to be unavailable. If neither ρ nor $\neg\rho$ the availability of the resource is unknown. We will then attempt to infer either ρ or $\neg\rho$ with the use of resource rules, but more on this in 2.1.6.

Resource scheme

To be able to form plans we need more than just atomic resources, some plans can require more elaborate structures than just a resource ρ. We therefore define $r \in \Theta(\mathcal{R})$ where $\Theta(\mathcal{R})$ is formed from the following grammar:

$$\begin{aligned} r &::= \rho | \bot | \top | \exists x : \rho(x) | \forall x : \rho(x) | \neg r | r \wedge r | r \vee r | r \rightarrow r \\ &\quad \textit{where } \rho \in \mathcal{R} \\ &\quad \textit{where } \rho(x) \in \mathcal{R} \end{aligned} \tag{2.1}$$

$\rho(x)$ is taken to mean a resource with parameters. An example of a sentence in the resource scheme could be: $\exists x, y : \texttt{free\_time}(x, y) \wedge y - x \geq$ 0:30. This says nothing about the concrete resource $\texttt{free\_time}$(1:30-2:00), just that there is a timeslot somewhere larger than half an hour, which may indeed be from

1:30-2:00, but may also be half an hour in the 'larger' resource `free time`(9:00-12:30). Furthermore we can use the standard inference rules for predicate logic to infer a sentence $r \in \Theta(\mathcal{R})$ from a set of resources $S \subseteq \mathcal{R}$. This will be useful in evaluating the availability of resources for plans.

2.1.4 Plans

We use an abstract idea of a plan: a plan $\pi \in \mathcal{P}$ will be said to *use* resources in $r_1, r_2, \ldots, r_n \subseteq \Theta(\mathcal{R})$ to *fulfill* desire $\delta \in \mathcal{D}$.

A plan may therefore be seen as: $\pi = r_1, r_2, \ldots, r_n \hookrightarrow \delta$. Where we define $\hookrightarrow$ to mean: *used by*. We will use this notation throughout the paper. Each resource scheme $r_1, \ldots, r_n$ is evaluated separately. r_1 are the *primary* resources used by the plan. Only if these are available will r_2 be evaluated and so forth. More on this in 2.1.5

We define the utility of plan:

$$\pi = r_1, r_2, \ldots, r_n \hookrightarrow \delta \; as : utility(\pi) = worth(\delta) - \sum_{i=1}^{n} \sum_{\rho \in r_i} cost(\rho) \qquad (2.2)$$

We define an agent's planbase as: $\mathcal{P}_A = \{\pi_1, \pi_2, \ldots, \pi_n\}$, where for each desire $\delta_i \in \mathcal{D}_A$ there is at least one plan $\pi_j \in \mathcal{P}_A$ that fulfills desire δ_i. We denote this relation between the plan and the desire it fulfills by $fulfills(\pi_j, \delta_i)$.

Why a staged approach to plans?

The reason for the staged resource sentences for plans is twofold. In the first place it lets us deal with a certain amount of uncertainty. We will only form an intention when the primary resources are available. This corresponds with a closed world view. If not all resources are available we don't care whether they are unavailable, or whether we just don't know about them. The other resources all must at the least be not unavailable. We can then check them in ascending order. The second reason arises from the fact that for this approach we need two groups, so why not expand it to any finite amount of groups of resources, corresponding to the ascending cost. It is time and bandwidth intensive to communicate with other agents and there is thus no need to do this if we know the intention cannot be formed in any case! In the case of a scheduling problem: an agent need not ascertain whether Peter, Paul and Mary have time for a lunch date if it knows its own user doesn't have time!

2.1.5 Intentions

An agent's intentions are the desires it is committed to fulfilling[7], using a specific plan. We therefore model intentions as a tuple:

Definition 2.2 ***Intentions***
An intention $\iota_i = \langle \delta_i, \pi_i \rangle$with $\delta_i \in \mathcal{D}_A$ and $\pi_i \in \mathcal{P}_A$, where $fulfills(\pi_i, \delta_i)$. An agent has an intentionbase $\mathcal{I}_A$ consisting of all its intentions ι_i.

Definition 2.3 ***Forming intentions***
Let $\iota = \langle \delta, \pi \rangle$ be an intention and $\pi = r_1, \ldots, r_n \hookrightarrow \delta$. The intention is $\mathbf{1-formed}$ *if all primary resources are available and none of the other resources are unavailable. In general: an intention is i-formed if $r_1, \ldots, r_i$ are all available and none of the resources in $r_{i+1}, \ldots, r_n$ are unavailable.*

Definition 2.4 ***Committing to intentions***
Let $\iota = \langle \delta, \pi \rangle$ be an intention with $\pi = r_1, \ldots, r_n \hookrightarrow \delta$ We incrementally i-form an intention for $i = 1 \ldots n$. We say an intention is ***fully-formed*** *if it is n-formed.*
Only fully-formed intentions may be committed to.

The 'committing to' of intentions is seen to be done in various stages, depending on the amount of secondary, tertiary, etc. resources that are needed. Only after an agent has specifically committed to an intention $\iota = \langle \delta, \pi \rangle$ and $\pi = r_1, \ldots, r_n \hookrightarrow \delta$ (In other words it is n-formed: $R \subseteq available(\mathcal{A}) \wedge R \vdash r_1, \ldots, r_n$) will the resources required for π be reserved (and thus removed from $available(\mathcal{A})$.

2.1.6 Resource rules

Because we are working in a mobile, dynamical environment, we need some way to 'discover' resources as we come across them. This will be done with resource rules.

Definition 2.5 ***Resource rule***
A resource rule γ is an expression of the form $\varphi \mapsto R$, where $\varphi \in \Theta(\mathcal{R} \cup \mathcal{B})$ is a constraint for the availability of resource tuple R. $\Theta(\mathcal{R} \cup \mathcal{B})$ is an extension of 2.1 in a straightforward way so it may also contain propositions over beliefs. We will call $Var(\gamma)$ the set of propositional variables in φ. An example is:
$\mathtt{ice\_cream\_stand(name, address)}$
$\wedge\ \mathtt{location(address, x_1, y_1)}\ \wedge\ \mathtt{location(me, x_2, y_2)}$
$\wedge\ \sqrt{(\mathtt{x_1} - \mathtt{x_2})^2 + (\mathtt{y_1} - \mathtt{y_2})^2)} \leq 200 \mapsto \langle \mathtt{ice\_cream(name, address)}, 0.6 \rangle$

An agent has a set of resource rules Γ_A, which it may use to infer the existence or changes to resources from beliefs. We define this new inference rule $\vdash_\Gamma$ as:

Definition 2.6 ***Resource inference***
$\vdash_\Gamma$ is a new inference rule which uses resource rules. Let $B \subseteq \mathcal{B}$, $\Re \subseteq \mathcal{R}$ and R a resource tuple then $B \cup \Re \vdash_\Gamma R$ means that there is a resource rule $\varphi \mapsto R \in \Gamma_A$ and $B \cup \Re \vdash \varphi$.

Consistency

To conserve consistency of the resourcebase under the above inference rule we impose the following restriction:
If $\gamma_1 = \varphi_1 \mapsto R_1, \gamma_2 = \varphi_2 \mapsto R_2$ such that $R_1 = \langle \rho_1, c_1 \rangle, R_2 = \langle \rho_2, c_2 \rangle$ with $\rho_1 \wedge \rho_2 \models \perp$ and $\gamma_1, \gamma_2 \in \Gamma_A$. Then $\varphi_1 \wedge \varphi_2 \models \perp$. This way it can never be so that γ_1 and γ_2 can be used in the construction of one and the same resourcebase.

Resourcebase construction

With the resource rules we can construct our resourcebase. Let $infer(\Psi, \Gamma) = \{R \in \mathcal{R} | \exists S \subseteq \Psi : S \vdash_\Gamma R\}$ The resourcebase is generated through the following inductive process:

```
Basic step: generate all resources $\mathcal{R}_0 = infer(\mathcal{B}, \Gamma_A)$
Loop: DO {
    $\mathcal{R}_i = infer(\mathcal{B} \cup \mathcal{R}_{i-1}, \Gamma_A)$
}
UNTIL $\mathcal{R}_i = \mathcal{R}_{i-1}$
$\mathcal{R}_A = \mathcal{R}_i$
```

We will abbreviate this process with $infer^*(\mathcal{B}, \Gamma_A) = \mathcal{R}_A$ with which we mean that $infer^*$ is the terminating execution of the above algorithm and $\mathcal{R}_A$ is the resulting resourcebase.

Lemma 2.7 *The algorithm does in fact terminate if* Γ_A *is a finite set.*

Proof. For each iteration i of the algorithm that does not terminate: $\mathcal{R}_i \supset \mathcal{R}_{i-1}$.
Therefore $\exists \rho \in \mathcal{R}_i : \rho \notin \mathcal{R}_{i-1}$ which we will call ξ.
Because $\mathcal{R}_i = infer(\mathcal{B} \cup \mathcal{R}_{i-1}, \Gamma_A)$ there must be a resource rule γ such that $\gamma \mapsto \xi \wedge Var(\gamma) \subseteq (\mathcal{B} \cup \mathcal{R}_{i-1})$.
Also because $\xi \notin \mathcal{R}_{i-1}$ this resource rule γ must have: $Var(\gamma) \not\subseteq (\mathcal{B} \cup \mathcal{R}_{i-2})$.
Ergo $Var(\gamma) \cap ((\mathcal{B} \cup \mathcal{R}_{i-1}) - (\mathcal{B} \cup \mathcal{R}_{i-2})) \neq \emptyset \Rightarrow Var(\gamma) \cap (\mathcal{R}_{i-1}) - \mathcal{R}_{i-2}) \neq \emptyset$.
Therefore for each step i of the algorithm there must be at least one resource rule such that: $\exists S \subseteq Var(\gamma) : S \subseteq \mathcal{R}_{i-1} \wedge S \not\subseteq \mathcal{R}_{i-2}$. Because Γ_A is not infinite, let $x = |\Gamma_A|$. Thus after x iterations there are no more resource rules for which the condition holds and the algorithm terminates. $\triangle$

We now see that if a resource is not in the resourcebase we do not know whether it is available or not (since all available resources will be added to the resourcebase explicitly with value true and all unavailable resources will be added explicitly with value false). Plan rules which require resources with truth value unknown cannot be executed until the resources are in fact known to be available.

2.1.7 The agent

We model the agent as being the following tuple:

Definition 2.8 ***Agent***
An agent $\mathcal{A} = \langle \mathcal{B}_A, \mathcal{D}_A, \mathcal{P}_A, \mathcal{I}_A, \mathcal{R}_A, \Gamma_A \rangle$, *where:*

- $\mathcal{B}_A$ *are the beliefs the agent has about the world.*
- $\mathcal{D}_A$ *are the desires the user has modeled for it.*

- *$\mathcal{P}_A$ are all the plans the agent may use to fulfill desires in $\mathcal{D}_A$.*
- *$\mathcal{I}_A$ are its intentions.*
- *$\mathcal{R}_A$ are resources available to the agent.*
- *Γ_A is the set of resource rules the agent has.*

Furthermore we define:

- *shared($\mathcal{R}_\mathcal{A}, \mathcal{B}$) the subset of resources of agent $\mathcal{A}$ shared with agent $\mathcal{B}$*
- *fulfillable($\mathcal{A}$) $= \{\delta \in \mathcal{D}_A | \exists \pi \in \mathcal{P}_A : (\delta, \pi) \in \mathcal{I}_A\}$*

2.2 Formalizing Context Changes

We now take a closer look at the three ways context may change as defined in section 1.2.

2.2.1 Information

The main problem with information is to discern *relevant* information from irrelevant. This problem is a spin-off from the frame problem [19]. For our problems we will define relevant information as: **information that changes resources that are used in plans**.

Because we defined our beliefbase as the depot for all information, a change in information is modeled as a change in the beliefbase.

Definition 2.9 *Let agent $\mathcal{A}$ have beliefbase $\mathcal{B}_A$ and $\beta \in \mathcal{B}$ be new information with certainty p.*

We have the following characteristics for adding new beliefs:

Consistency *If $\mathcal{B}_A \cup \{\beta\} \models \bot$ then we speak of conflicting beliefs. In the case of conflicting beliefs, some form of belief revision must take place.*

Non-redundancy *If $\exists(\beta, p') \in \mathcal{B}_A$ we update the beliefbase so that it contains $(\beta, \max(p, p'))$.*

Other cases *In all other cases $\mathcal{A}$'s beliefbase becomes $\mathcal{B}'_A = \mathcal{B}_A \cup \{(\beta, p)\}$*

With this definition of beliefs, we can model contextual relevance of the information.

Definition 2.10 ***Relevance***

We say a change in beliefs is relevant to agent $\mathcal{A}$ if and only if the new beliefbase $\mathcal{B}'_A$, resulting after an update, affects the resourcebase $\mathcal{R}_A$:

Let $\mathcal{B}_A$ be $\mathcal{A}$'s old beliefbase and $\mathcal{R}_A$ the resourcebase constructed from it. Let $\mathcal{B}'_A$ be $\mathcal{A}$'s new beliefbase and $\mathcal{R}'_A$ be the new resourcebase constructed from *it. We say the change in beliefs ΔB is relevant if $\mathcal{R}_A \neq \mathcal{R}'_A$*

We say the change was positive *if $\mathcal{R}_A \subset \mathcal{R}'_A$ and* negative *if $\mathcal{R}_A \supset \mathcal{R}'_A$.*

And so all information relevant to impromptu coordination will by definition culminate in a change in the agent's resourcebase.

Belief Revision

As mentioned in definition 2.9, new information may lead to inconsistencies in the beliefbase. Simply adding new information to the beliefbase is therefore not allowed. After updating the beliefbase, it must still comply with both the outer and inner consistency rules. We allow the beliefbase to be a union of various different datastructures and consistency is handled differently for each of them. Many of the used beliefbases will have the structure of a relational database and consistency of databases is a well researched topic. For further reading on updating databases with incomplete information we refer to Marianne Winslett's book [36]. For simpler datastructures consisting of only literals the following rule is sufficient:

Postulate 2.11 *Let $B_a \in \mathcal{B}_A$ be a representation in the beliefbase of agent $\mathcal{A}$ and $\beta \in \mathcal{B}$ be new information with certainty p. β conflicts with B_a iff: $\exists(\beta', p') \in B_a : \beta' = \neg\beta$. We use the following rule to resolve the conflict: $p \geq p' \Rightarrow B'_a = \Big(B_a - \{(\beta', p')\}\Big) \cup \{(\beta, p)\}$. Where B'_a is the new representation in $\mathcal{A}$'s beliefbase. Otherwise the beliefbase remains unchanged and the new information is rejected.*

This may be used in such structures as a calendar or GPS location-base, where information can be considered as atomic literals.

For a more complete overview of the various forms of belief revision see [10].

2.2.2 Plans

Own plans

Changing your own plans will usually create a change in resources, so changes in your own plans may be seen as a sort of recursive change in your own resources. On a lower level these changes may therefore be treated equally, but for a better understanding we group them separately.

We discern two cases. We may already intend to fulfill a desire δ, using plan π, but a new plan π' may supersede π if:

Definition 2.12 ***Supersedence***
A plan $\pi = r_1 \ldots r_n \hookrightarrow \delta$ is said to be superseded by a plan $\pi' = r'_1 \ldots r'_m \hookrightarrow \delta'$ if and only if $\delta = \delta'$ and utility$(\pi') \geq$ utility(π) Notation: $\pi' \succcurlyeq \pi$

The trivial case where $\pi' = \pi$ is ignored. We deal with supersedence by removing (δ, π) from our intentionbase and replace it with (δ, π') creating a change in $available(\mathcal{A})$. This we may handle as a case of changing resources.

The second case we discern is conflicting plans. A plan π' conflicts with plan π if:

Definition 2.13 ***Conflict of plans***
$\pi = r_1 \dots r_m \hookrightarrow \delta$ is said to conflict with the collection of plans $\pi_1 = r_{11} \dots r_{1k} \hookrightarrow \delta_1, \dots, \pi_n = r_{n1} \dots r_{nj} \hookrightarrow \delta_n$ if:
Let $R = Var(\{r_1, \dots, r_m\})$ then π conflicts when $R \cap (\bigcup_{i \leq n} R_i) \neq \emptyset$ and π doesn't supersede any of the plans π_i.

The resulting conflict must be resolved in some way and it will be up to the agent to decide which combination of plans is the most efficient. We update $\mathcal{I}_A$ accordingly creating a change in $available(\mathcal{A})$ and handle the rest as a change in resources. Because a plan always supersedes itself (albeit in a trivial manner), it never conflicts with itself (which would be silly).

Other agents' plans

A completely different situation is if other agents' plans change. If this affects our own plans, we fall back into the previous category, but there are situations in which another agents' plans don't affect any of our current plans, but have an effect on the availability of shared resources. For example: a friend decides to go to the beach tomorrow, which opens up the opportunity for you to go to the beach with him. These changes will, however, be communicated, as there is otherwise no way of knowing another agents' plans (barring severe breaches of privacy). They will be treated as a change in information, along with other communication acts.

2.2.3 Resources

We model resources as an open world and therefore with 2 types of negation. When a resource is known to be available it will be in the resourcebase. The resourcebase may also contain resources known *not* to be available by having strong negation in the resourcebase. Others we won't know anything about and will therefore not be included in the resourcebase and assumed to be unavailable until required, at which point we will need to verify the availability of a resource. This can be triggered by plans' staged construction. Let $\pi = r_1, r2 \hookrightarrow \delta$ be a plan. The first stage will only be triggered by the primary resource scheme r_1 and only if $r_1 \not\models \bot$. If r_1 succeeds we start checking the resources required for r_2. These will require new information, because otherwise they would have been inferred already from the beliefbase by resource rules. This may be information from sensors, online databases, webpages or other static sources or by communicating with other agents.

2.3 Formalizing impromptu coordination

Now that we have shown how all contextual changes eventually map to resource changes, we can show how resource changes affect our agent. We will formalize the idea of recognition of opportunities in the following manner:

Definition 2.14 ***Opportunity***
Let agent $\mathcal{A}$ have desire $\delta \notin fulfillable(\mathcal{A})$. We define opportunity as a change in $\mathcal{R}_A$ such that an i-formed intention $\iota = (\delta, \pi)$ may become j-formed, with $j > i$.

Definition 2.15 ***Successful opportunity***
A successful opportunity is an opportunity such that intention ι is fully-formed and committed to.

2.3.1 On opportunity

An obvious question is: why do we not use a more straightforward definition for opportunity? That is, one that doesn't require definitions of formedness and commitment, but instead is given in terms of how the intentionbase may be changed to maximize either the utility of all plans or worth of all desires. The answer is twofold:

Firstly we wish to stay committed to plans we had committed to execute. This from both a consistency viewpoint and more importantly an HCI viewpoint. The problem is to support impromptu coordination and each intention that has been committed to, in practice, will have been approved by a user. This means that we cannot just take intentions out of our intentionbase, this requires permission from the user. This would be an illogical use of our agent, we would rather have the user remove intentions (by removing the desire from the desirebase). The only situation in which the agent should remove an intention is if the plan has been executed, or else if the necessary resources become unavailable.

Secondly we are dealing with impromptu coordination: a scenario in which opportunity is intuitively described as "being able to fulfill a desire that wasn't possible before", not to restructure all your plans so as to optimize your efficiency; that is called planning!

2.3.2 Concluding remarks

Now that we have given definitions of an agent structure for supporting impromptu coordination and defined formally how impromptu coordination works we may go on to forming an architecture for this agent in the next section. Note that beliefs are not modeled in any of the normally used logic systems for agents, such as KD45. Although the possibility is left open to do so, we don't expect an implementation to do so, because it makes more sense to use the datastructures that are already being used for the specific contexts. This does, however, leave the enforcement of outer consistency up to the final implementation. Further research could be done into the integration of the outer consistency rule into the logical framework using modal logic[20].

2.4 A concrete example

We will show how this framework can be applied to one of the examples in chapter 1; we will show how we can formalize the process of buying a book. For the sake of simplicity we will only formalize the basic situation, but the more complex example with multiple contexts being put together can be formalized in the same manner. We compare prices and delivery times of online and offline book vendors. We also use information about the location of the offline vendors and the agent's own location.

The scenario: *You wish to buy a book and know of various different book stores. First you phone the local bookstore, who are redoing their inventory system and don't know whether the book is in stock, but can give you a price. You have an appointment in the afternoon on the wrong side of town, so you check nile.com and stablesandbase.com, which are online bookstores. You find they are cheaper than the local bookstore and have a delivery time of 5 days. You would prefer to have it sooner than that, but the express delivery rate is significantly more expensive. Before placing the actual order, your wife calls to say she has an important meeting and needs you to pick up the children from school. This takes you near the bookstore, who did indeed receive the book in the afternoon shipment and you can pick it up.*

The details: your beliefbase consists of the prices of books from various online and offline stores. It also contains the delivery time for each of these. The locations beliefbase contains information about your own location as well as the location of all the offline bookstores in question.

- sale(nile, book, 30, online)
- sale(localstore, book, 33, offline)
- sale(stablesandbase, book, 32, online)
- position(localstore, 10, 12)
- position(me, 1, 1)
- deliverytime(nile, 6)
- deliverytime(stablesandbase, 3)

Your desirebase contains the desire to buy a book: <get(book), 50>.

You have resource rules:

$$\begin{aligned}
&\mathit{position}(\mathit{location}, x, y) \wedge \mathit{position}(\mathit{me}, x2, y2)\\
&\qquad \mapsto < \mathit{distance}(\mathit{location}, \sqrt{(x - x2)^2 + (y - y2)^2}), \sqrt{(x - x2)^2 + (y - y2)^2} >\\
&\mathit{sale}(\mathit{store}, \mathit{item}, \mathit{price}, \texttt{online}) \wedge \mathit{deliverytime}(\mathit{store}, T)\\
&\qquad \mapsto < \mathit{available}(\mathit{store}, \mathit{item}, \texttt{online}), \mathit{price} + T >\\
&\mathit{sale}(\mathit{store}, \mathit{item}, \mathit{price}, \texttt{offline}) \wedge \mathit{distance}(\mathit{store}, T)\\
&\qquad \mapsto < \mathit{available}(\mathit{store}, \mathit{item}, \texttt{offline}), \mathit{price} + T >
\end{aligned}$$

We now have the plan:

$$\begin{aligned}
\pi_1 =& \exists \mathit{store}, \mathit{item} :< \mathit{available}(\mathit{store}, \mathit{item}, \mathit{method}), \mathit{cost} > \wedge\\
& \forall \mathit{store}2, \mathit{item}2 :< \mathit{available}(\mathit{store}2, \mathit{item}2, \mathit{method}), \mathit{cost}2 > \wedge \mathit{cost} < \mathit{cost}2\\
& \hookrightarrow \mathit{get}(\mathit{item})
\end{aligned}$$

Initially we form the intention:

$$\left\langle \mathit{available}(\texttt{stablesandbase}, \texttt{book}, \texttt{online}) \hookrightarrow \mathit{get}(\texttt{book}), 15 \right\rangle$$

However, if we move around and come closer to the store the belief position(me, X, Y) will change. If for instance we come near the local bookstore our belief might become position(me, 9, 13).

In that case the resource distance(localstore, D) will have changed from 14.5 to 1.4 and thus the cost of available(localstore, book, offline) will have changed accordingly.

We therefore see we form a new instance of plan π_1, which supersedes our previous intention:

$$\left\langle \mathit{available}(\texttt{localstore}, \texttt{book}, \texttt{offline}) \hookrightarrow \mathit{get}(\texttt{book}), 15.6 \right\rangle$$

So unless the user had already given the go ahead to commit the previous intention, we can still switch to forming this intention and ask the user to give his okay for committing to it (in this case quickly walking to the bookstore. In the former case, filling in his credit card information for the online store).

Obviously this is a very much simplified example. The costs for the resources are too simplistic and there are only 2 contexts being used. We could, for instance, have used the calendar context to realize there was an appointment for the afternoon in town, which would bring us near the local bookstore. Then we would possibly have never formed the intention to buy the book online. We hope, however, that this example shows how the framework is supposed to work in a dynamic environment.

Chapter 3

A framework enabling impromptu coordination

Mobile Technology is developing extremely fast and many different forms of connectivity are developing side by side with more and more different mobile devices. We will assume a permanently connected agent, although our architecture doesn't depend on it.

Just as O'Grady and O'Hare in [22], we feel the best way to accomplish the robust- and adaptiveness necessary in a dynamic environment is best achieved by using a Multi-Agent System. However, unlike them, we have most of our agents run on the mobile device itself. To achieve this, we need to develop fairly small and simple algorithms. In chapter 2 we discern between resources and beliefs as well as between planning rules and resource rules. This enables us to split the computation the agent has to do over different asynchronous parts of the system. It is virtually analogous to robot architectures, where raw sensor data is pre-processed and planning is done using this pre-processed world model. It results in abstract actions, which are then processed and turned into concrete actions [3], just as our plans are abstractions from real-world plans, which the intention-agent executes.

3.1 A Multi-Agent approach of impromptu coordination

We define 3 different kinds of agent in our system. Firstly we may have an arbitrary number of context-agents. They search for information from whatever context they are responsible for and send this information on to the so-called 'brain'-agent. This process incorporates the resource-rules in 2.1.6 and also checks for the relevance of the resulting resources. The context-agents don't have to run on the PDA, but may be running on any computer where the contextual information is stored. Some may run on the PDA, but others may be running on the user's home computer or a server elsewhere. The system as a whole will model the agent described in chapter 2, while each agent will fulfill its own role within the system autonomously. The brain-agent does the brunt of the computation, while the context-agents each supply the brain-agent with knowledge of the availability of resources. Based on these resources, the brain-agent may then recognize opportunities to fulfill its desires and send the intention-agent the go-ahead to commit to certain plans. To return to the robot analogy: the system's sensors are the context-agents, while the system's cognition is handled in the brain-agent. That just leaves the actors, which are implemented as the intention-agent. Together the system will support the user with impromptu coordination.

3.2 A formal MAS

We will follow the logical structure of the agent as outlined in 2.1, now giving an overview of the architecture needed to model these structures in software agents.

3.2.1 Beliefs - The Context Information System

We will model an agent's beliefbase as a representation of the specific contexts given in chapter 1.1.1 that we wish the agent to be aware of. In 2.1.1 we announced different beliefs could possibly be represented differently, depending on the easiest form to represent it, as well as the capabilities on a PDA. We will take this concept slightly further, modeling each different context in a different context-agent. Each context-agent's beliefbase contains the necessary information on a context.

Inner consistency rule Every agent is responsible for keeping its own beliefs consistent, whatever representation is used. Relational databases have sophisticated methods in place to guarantee structural consistency [31] and so do beliefbases based on predicate logic. A calendar could be said to be consistent by requiring it to have only one appointment per person at any given time. For any representational mode some type of consistency rule must be devised. Alternatively the agent itself may be used to keep the representation consistent by revising its beliefbase. Maintaining a consistent beliefbase is an ongoing theme of research, which we won't discuss here, except to say that it is possible [1] in a variety of ways.

Outer consistency rule A problem arises for the outer consistency rule, because each representation is maintained by a different agent. We therefore require complete disjunction of functionality. There can be no syntactic inconsistency if we require complete disjunction: if context agent $\mathcal{C}_1$ believes ϕ, there is no context-agent $\mathcal{C}_i$ with beliefbase B_i such that $B_i \cup \phi \vdash \bot$, simply because B_i must have no beliefs relating to ϕ. Each context-agent may only contain beliefs about the contexts it is in charge of. The context-agent in charge of 'location' can therefore not have information about 'time'. Semantic inconsistency is still a problem, but not one that can be solved through formal logical rules. If φ represents the belief that 'it is raining' and ψ represents the belief that 'it is not raining', they are semantically inconsistent, yet not syntactically. This is something that must be dealt with at the application level, or in our case, during resource generation.

3.2.2 Desires

Desires are maintained by the brain-agent. They are modeled exactly as described in 2.1.2, as a tuple of the desire itself, its worth and a desire type (one-off, recurring or continuous). They ultimately give the MAS a purpose: to fulfill as many desires as possible under the given restraints.

3.2.3 Resources

The brain-agent's beliefs are what we named resources in 2.1.3. They are modeled as atomic propositions, with a cost, just as described in definition 2.1. Usually the brain-agent won't update the resources itself, but context-agents will do that with resource rules, further described in 3.2.6. Resources are stored with a 3-value logic: true, false and unknown. If a resource is known to be available it will be stored as true, but we also wish to keep track of resources we know to definitely be unavailable, so we'll store them as being false. Everything not stored explicitly is modeled as unknown. It may be a context-agent has information about them, but as long as it is not explicitly mentioned in the resourcebase it will be treated as unknown.

The cost of a resource will be kept up to date in the resourcebase and so will the availability. In addition a resource may be 'reserved'. We discern 2 different kinds of resources: 'usable' and 'persistent' resources. Usable resources can be 'used up', such as time certain commodities. Persistent resources cannot be used, but may more accurately be described as 'requirements' for a plan to succeed, such as, for example, being near an ice cream stall. We are required to be at a certain place for our plan to succeed, however our location is a 'requirement' and cannot be 'used up'. Other plans requiring us to be in the same place may still use this resource.

The reason we group these two kinds of resources together is that for the forming of intentions the distinction between persistent and usable resources is irrelevant. They are all required to be available for the plan to succeed. However when committing to intended plans the way they are treated is different. Also for the construction of our resourcebase these two different kinds of resources can be treated exactly the same, so although they are conceptually quite different they can be treated in similar ways except for when they are 'used'. When an intention is committed to, the resources it requires are flagged as being 'reserved'. Once the plan has been executed, the usable resources will have been used, and hence are unavailable, while persistent resources cannot become unavailable by using them. To ensure the integrity of our intentionbase we therefore treat reserved usable resources as unavailable, while treating reserved persistent resources as available.

This same mechanism may also be used to alert the agent to resources being changed for plans that are already committed to. If a resource's availability is changed while it is reserved, some commitments may no longer be executable. We therefore use the reserved flag to also know when intentions need to be rolled back, due to a changed situation.

3.2.4 Plans

Plans in the brain-agent are a sorted set of logical sentences in the resource scheme described in equation 2.1 together with the desire they fulfill. A plan is an abstract representation of a more concrete plan in the intention-agent. The intention-agent will contain plans in the more traditional BDI sense; with actions and subgoals to achieve. It is responsible for executing these plans when the brain-agent says so. It generates the more abstract plans by finding the resources required to complete the actions in its plans and passes these plans on to the brain-agent, which can use them to check the availability and calculate which plans to execute.

Utility

The utility of a plan is calculated based on the maximum cost of the required resources. Due to the changing costs of resources, it is sometimes necessary to recalculate the utilities. If the cost of a resource is not yet accurately known we choose the worst-case scenario cost, so the utility may not always be accurate, but always errs on the low side. We guarantee that the actual utility is higher or equal to the calculated utility. However, it should always be our aim to get information as accurately, so as to have good estimates of the utility of plans (and therefore which plans to commit to).

3.2.5 Intentions

In the brain-agent we do not really distinguish between plans and intentions. Intentions are simply plans with an i-formedness, and information about whether or not they are committed to yet. In the logical description we described them separately, but in practice they will always be used together, so we don't treat them separately. We don't discern intentions from plans except by their i-formedness.

i-forming plans

Our staged approach to plans is reflected in the i-formedness of a plan. Being i-formed means that the first i entries in the plan's array of resources are evaluated to true, while the last $n - i$ do not evaluate to false (may be either true or unknown). When a plan is n-formed and the user commits to it, the plan is switched to committed, the resources are reserved and the intention-agent is notified that it may start execution of the plan.

3.2.6 Resource Rules

Resource rules are implemented as plans of the context-agents. Their aim is to supply the brain-agent with relevant resources, which they can do by following plans to distill resources from their contexts. The resource rules of 2.1.6 provide the means with which to do so, inferring the resources from the beliefs. However, we defined resource rules as being able to depend on multiple contexts. This creates a problem: for consistency reasons each agent may only have beliefs about its own context, but it must also know things about other contexts if resource rules are to be implemented properly. Superficially it would seem that these two premises cannot be unified. However, this can in fact be achieved in the following way: with each context only being maintained by one single agent, we have already centralized the beliefs concerning these contexts. Resources also fall within these contexts, so we can limit which resources are updated by which agents. Thus we can define different resource contexts, each only updated by 1 context-agent. Each context-agent may be in charge of more than one resource context, but the reverse is never true.

This still doesn't solve resource rules spanning multiple contexts, but because we are free to design our own agent network, we will do so in such a way that there are few resources that depend on rules spanning multiple different contexts. If these do need checking, we can rewrite the resource rule as a planning rule. Instead of having a simple resource with a complicated resource rule, we can require many resources in a complicated planning rule. That way each resource's availability is only dependent on local information and the availability can be verified from the beliefs of a context-agent.

Combining contexts - plan rules vs. resource rules

As said, resource rules may only draw upon the beliefs available in one single context-agent. This makes updating the resourcebase a fairly easy process when it comes to resources depending on only one context, but in some situations it seems that to decide on a resource's availability we depend on more than one context. Displacing this dependency to the planning rules may seem like a 'cheap trick', rather than a true solution. Don't these problems just come back to bite us when we try to form the plans?

Luckily this is not the case, due to the way our system works. The resourcebase is the brain-agent's beliefbase and as such must be kept consistent. That is helped by the restriction given in section 2.5, however enforcing this restriction in a multi-agent system is far from trivial. With the beliefs distributed over various context-agents it would require a lot of communication to check this rule for rules spanning multiple contexts. In addition, it'd require a lot of overhead and each context-agent would not only need to know which of its beliefs are relevant for its own resource rules, but also which could be relevant for another agent's resource rules. This also bears the risk of creating deadlocks. Context-agents are each waiting for an answer from each other to validate the availability of a resource, which is in turn needed to reply to the other context-agent. All this overhead could be solved by moving the computation of resource rules and resourcebase construction to the brain-agent. However, this would lose the advantage gained by distributing them: context-agents can run anywhere (not just on the mobile device) and only communicate relevant information to the brain-agent when necessary. Restricting resource rules to one single context solves this computational problem, letting each agent act in relative seclusion. Consistency is guaranteed by the restriction for resource rules, which is relatively simple to implement on a single agent. We don't run into problems at a later stage due to the complete disjunction of contexts, also for resources. Planning rules don't have this problem with consistency, because they are not meant to generate any kind of beliefbase. There is one caveat: the generation of planning rules becomes more difficult, because the resources themselves are more basic.

An example

We could model the availability of a book with a resource rule as we did in 2.4:

$$\begin{aligned}&\mathit{sale}(\mathit{store}, \mathit{item}, \mathit{price}, \texttt{offline}) \wedge \mathit{distance}(\mathit{store}, T)\\ &\qquad\qquad \mapsto < \mathit{available}(\mathit{store}, \mathit{item}, \texttt{offline}), \mathit{price} + T >\end{aligned}$$

spanning 2 contexts: location and online information (although the bookstore is a real shop, the information about the book's sale is still obtained online). This is not allowed in our framework. We thus form 2 different resource rules:

$$\begin{aligned}&\mathit{location}(x, y) \wedge \mathit{me}(x2, y2)\\ &\qquad \mapsto < \mathit{distance}(\mathit{location}, \sqrt{(x - x2)^2 + (y - y2)^2}), \sqrt{(x - x2)^2 + (y - y2)^2} >\end{aligned}$$

$$\begin{aligned}&\mathit{store}, \mathit{item} : \mathit{sale}(\mathit{store}, \mathit{item}, \mathit{price}, \mathit{stock}, \texttt{offline})\\ &\qquad \mapsto < \mathit{info\_available}(\mathit{store}, \mathit{item}, \texttt{offline}), \mathit{price} + T >\end{aligned}$$

each evaluated by a different context-agent and communicated to the brain-agent. The planning rule then becomes:

$$\begin{aligned}\pi_1 = &\exists \mathit{store}, \mathit{item} :< \mathit{info\_available}(\mathit{store}, \mathit{item}, \texttt{offline}), \mathit{cost} > \wedge\ \mathit{distance}(\mathit{store}, X)\ \wedge\\ &\big(\forall \mathit{store2}, \mathit{item2} :< \mathit{info\_available}(\mathit{store2}, \mathit{item2}, \texttt{offline}), \mathit{cost2} > \wedge\ \mathit{distance}(\mathit{store2}, X_2)\\ &\wedge (\mathit{cost} + X) < (\mathit{cost2} + X_2)\big)\\ &\hookrightarrow \mathit{get}(\mathit{item})\end{aligned}$$

which only deals with other offline stores and is already more complicated than the 'old' planning rule:

$$\begin{aligned}\pi_1 = &\exists \mathit{store}, \mathit{item} :< \mathit{available}(\mathit{store}, \mathit{item}, \mathit{method}), \mathit{cost} > \wedge\\ &\forall \mathit{store2}, \mathit{item2} :< \mathit{available}(\mathit{store2}, \mathit{item2}, \mathit{method}), \mathit{cost2} > \wedge\ \mathit{cost} < \mathit{cost2}\\ &\hookrightarrow \mathit{get}(\mathit{item})\end{aligned}$$

So, although we don't get rid of any of the complexity of dealing with context-spanning resources (which we just moved), we do solve problems regarding their mutual consistency.

3.2.7 The system

Our entire framework is now comprised of:

- 1 Brain-agent $\texttt{B} = \langle \mathcal{R}_A, \mathcal{I}_A, \mathcal{D}_A \rangle$
- 1 Intention-agent $\texttt{I} = \langle \mathcal{P}_A \rangle$
- * Context-agents $\texttt{C}_1, \ldots, \texttt{C}_n$, with each $\texttt{C}_i = \langle \mathcal{B}_{A_i}, \Gamma_{A_i} \rangle$

where each construct is as described in definition 2.8. The only thing different is that the complex agent described there is split up into multiple smaller agents communicating where needed so the system encompasses the same structures as the single agent. However, for it to work as intended more is needed than merely the structure described. The power of this split up system is in the way it handles changes with the deliberation cycle.

3.3 Deliberating with context changes

3.3.1 Context-agents

Each context-agent will be in charge of updating the brain-agent with information *relevant* to the user. We defined this in 2.2.1 to be that information that changes resources. Each context-agent needs to be aware of which resources are needed. It need only know about those resources that fall within its own context. It also need only communicate back to the brain-agent when some of its beliefs change the status of this resource.

If the resource changes, a message will be sent to the brain-agent notifying it of this fact. It can then change its resourcebase and activate the deliberation cycle.

Resourcebase construction

In section 2.1.6 we gave an algorithm for the construction of resources. However, this generates the entire resourcebase, which may not always be needed. We only need to construct that part of the resourcebase that is relevant. Although we initially described relevant information as that pertaining to changes in any resources, we can, with impunity, narrow this definition further down:

Definition 3.1 ***Practical relevance:*** *information is relevant if it pertains to changes in resources needed for at least one plan π such that $\exists \delta \in \mathcal{D}_A : fulfills(\pi, \delta)$. In other words, information is relevant if it affects our intentions.*

This way we can keep the resourcebase small in such a way that it only contains those resources necessary for the functioning of the brain-agent.

We must propagate the relevance of resources to the context-agents. If a resource is marked as relevant to a context-agent, it means that:

- There is a plan in the brain-agent that requires that resource
- The resource falls within the context-agent's context

The resourcebase can now be constructed, by following the rules for evaluating the resources which are labeled relevant. The resourcebase is then constructed in the opposite manner to the algorithm described earlier. We evaluate the resource rules for the resources labeled relevant. If these resource rules require other resources to be available, the resource rules to evaluate those resources are checked, etc. The algorithm can be described as follows:

```
Basic step:
Select the set of resources $S_0$ which are relevant to our plans.
$R_0 := \emptyset$
FOREACH $r \in S_0$ DO {
    IF $\exists \gamma \in \Gamma_A : \gamma = \phi \mapsto r \wedge \Theta(\mathcal{B}_A) \vdash \phi)$
        THEN $R_0 := R_0 \cup \{r\}$
}
Loop:
DO {
    Select the set of resources $S_i$ which are relevant.
    $R_i := R_{i-1}$
    FOREACH $r \in S_i - R_i$ DO {
        IF $\exists \gamma \in \Gamma_A : \gamma = \phi \mapsto r \wedge \Theta(\mathcal{B}_A \cup \ R_{i-1}) \vdash \phi$
            THEN $R_i := R_i \cup \{r\}$
    }
}
UNTIL $S_i = S_{i-1}$
$\mathcal{R}'_A = R_i$
```

For easier reference we will call the foreach loop in the above algorithm $generate(S, \Gamma_A, \Psi)$ in analogy to the $infer(S, \Psi)$. It then stands to reason that we call $\mathcal{R}'_A = generate^*(S_0, \Gamma_A, \mathcal{B}_A)$ the complete execution of the algorithm with $\mathcal{S}_A$ the set of all eventually relevant resources.

Theorem 3.2 *The resourcebases resulting from $infer^*(\mathcal{B}_A, \Gamma_A)$ and $generate^*(S_0, \Gamma_A, \mathcal{B}_A)$ are equivalent if all resources are said to be relevant.*

Proof. Let r be a resource in $\mathcal{R}_A$ where $\mathcal{R}_A = infer^*(\mathcal{B}_A, \Gamma_A)$ generated using the algorithm in 2.1.6. Let $\mathcal{R}_i$ be the resourcebase where r was first added, then

$r \in infer(\mathcal{R}_{i-1} \cup \mathcal{B}_A, \Gamma_A)$ and $\gamma \in \Gamma_A, \gamma = \varphi \mapsto r$, such that $\Theta(\mathcal{R}_\mathcal{I} \cup \mathcal{B}_A) \vdash_\Gamma \psi$ with $j \in [0, i) \subset \mathbb{N}$. We can recursively regress through the algorithm back to step 1: $\mathcal{R}_1 = infer(\mathcal{B}, \Gamma_A)$ Let V be all resources needed in any resource rule to eventually generate r and let all resources in V be marked relevant, then $V \subseteq \mathcal{S}_A$ and because we have generated all resources in V we know that for each $v \in V : \exists \gamma \in \Gamma_A : \gamma = \varphi \mapsto v$ and thus $V \subseteq \mathcal{R}'_A$. More specifically $r \in V \Rightarrow r \in \mathcal{R}'_A$. Thus if all resources are marked as relevant then every resource generated by $infer^*(\mathcal{B}_A, \Gamma_A)$ will be in the resourcebase generated above too and $\mathcal{R}_A \subseteq \mathcal{R}'_A$.

Let r be a resource in $\mathcal{R}'_A = generate^*(S_0, \Gamma_A, \mathcal{B}_A)$ generated in the manner above. Then r is relevant and there is a resource rule $\gamma \in \Gamma_A$ such that $\gamma = \varphi \mapsto r$ and $Var(\gamma) \subset \mathcal{R}'_A \cup \mathcal{B}_A)$. Let $V = Var(\gamma) \cap \mathcal{R}$ then $V \subseteq \mathcal{S}_A)$ Therefore each resource rule to generate the resources in V will have been evaluated. Let V' be the set of resources required in all the resource rules to generate V. Then $V' \subseteq S_A$ as well. Recursively generate the resources required for generating r until $V^* = \emptyset$. This can be done and follows from lemma 2.7. Let G be the resource rules used in the last step of generating V^*. Because $V^* = \emptyset$ it follows that $\bigcup_{\gamma \in G} Var(\gamma) \subset \mathcal{B}_A)$. So these same resources could be constructed using $infer^*(\mathcal{B}_A, \Gamma_A)$. Thus $\mathcal{R}'_A \subseteq \mathcal{R}_A$.

Therefore $\mathcal{R}_A = \mathcal{R}'_A$ $\triangle$

If not all resources are marked relevant the first part of the proof doesn't hold and we get $generate^*(S_0, \Gamma_A, \mathcal{B}_A) \subseteq infer^*(\mathcal{B}_A, \Gamma_A)$, which is the way in which our resourcebase generally will be encountered in practice.

3.3.2 Brain-agent

We have given an outline of the context-agents, how the resourcebase is constructed, how the overall system can reason and what the brain-agent's part is. Now it's time to describe the deliberation cycle. All deliberation is event driven, so we will discuss the deliberation cycle in certain scenarios. However to guarantee the consistency of an agent there must be certain sequences of deliberation that have to be treated as 'building blocks' for the deliberation.

Initialization

To start with the system must be in a stable, consistent state. This is achieved with algorithm 3.1.

This algorithm guarantees that we have a set of intentions that maximizes the utility our agent can achieve under the currently available resources.

A closer look at relevance

Due to the staged approach of plans it is not necessary for all the resources required by all plans to be relevant from the beginning. The reason we have a

```
the brain-agent submits its desires to the intention-agent
the intention-agent returns all plans to fulfill said desires
the brain-agent calculates a set S of plans that maximize the overall utility,
   while remaining consistent (none of the plans conflict (2.13))
repeat until either all plans have been evaluated or
             we have a complete set of n-formed plans {
   for each plan in this set {
      for i = 1 to n
         check if resources are available to i-form the plan
   }
   recalculate utilities for all plans and check whether the set is still optimal
   if not all the plans in the set are n-formed
      attempt to find a new set that maximizes the overall utility
}
commit all n-formed plans in the maximal set
reserve resources of committed plans
```

Algorithm 3.1: INITIALIZATION

staged approach is to ensure that the more costly resource checking is done only when the availability of easily checked resources has been confirmed. If these are unavailable it is irrelevant whether the later ones are available. Thus we will incrementally add these to the list of relevant resources. If a desire is fulfilled (a plan to fulfill it has been executed), all plans that fulfill that desire can be removed and the resources are no longer relevant for those plans. More on this in 3.3.2.

Resource checking

In the previous section we said rather simply "check if the resources are available". However, if the resource isn't already known in the resourcebase, this is not quite so simple. The following steps must be taken:

- The brain-agent requests the availability of a resource from the corresponding context-agent.
- This context-agent must use its resource rules to check the availability.
- The context-agent informs the brain-agent whether the resource is available or not.

We already restricted resource rules to a single context in 3.2.6, so for each resource at most one query to one context agent is required.

```
for all plans requiring the changed resource do {
   if the plan is committed
      reevaluate all resources. If this fails run the algorithm for intention rollback (3.4)
   else with an i-formed plan: for j is 1 to i do {
      check resources for the jth resource set
      if unavailable set plan to 0-formed
      if unknown set plan to j-1-formed
   }
}
```

Algorithm 3.2: Negative change in resources

Deliberation cycle

After initializing an agent as above we have a stable situation. Some plans will have been committed to and their resources reserved, others will still be 0-formed, while still others will be somewhere in the middle of the forming process. Due to the event driven nature of our architecture this will stay the same until something happens. We have a 2 different changes:

- Context changes
- Desire changes

We will further split up the context changes in the same manner as in 2.2 into a couple of different events that may occur. These will each be handled in their own way, which we will see in the following sections.

Changes to relevant information

If a context-agent's beliefbase changes it will check whether anything changes in the relevant resources. If so, it'll send a message to the brain-agent. The brain-agent will receive this and reevaluate the plans that require this resource. In case of a positive change it may (and often will) advance the formation of a plan, whereas a negative change will lower the 'formedness' of a plan, possibly even of an n-formed or in the worst case a committed plan.

The case of a resource for a committed plan becoming unavailable in 3.2 is fairly rare, however we need to take it into account. Also having more than one intention become n-formed in 3.3 is fairly rare, but if it was a very critical resource it may be. In that case a maximal set must be calculated. If the resource can only be used once it will be only 1 intention, but if it cannot be 'used up' we may be able to commit to all the plans and thus fulfill many desires.

```
for all i-formed plans where i != n, requiring the changed resource do {
   for j is i+1 to n do {
      check resources for the jth resource set. If available set plan to j-formed
   }
   recalculate utility of plan
}
select a maximal set from any n-formed plans
commit all n-formed plans in the maximal set
reserve resources of committed plans
```

Algorithm 3.3: Positive change in resources

Changes to plans and intentions

If a new plan is added, it needs to be evaluated. This is fairly straightforward: if it is for an unfulfilled desire, check the resources, i-form the plan and if i=n commit to it and reserve the resources. The more interesting plan change is when commitments are undone. This is done when the intention-agent notifies the brain-agent that a plan has been executed. It also happens in the rare situation above, if a resource for a committed intention becomes unavailable. In the case of the removal of a plan or uncommitting of an intention, we run 3.4. In addition, if the plan wasn't removed, but had to be rolled back for other reasons (unavailable resources), the formedness and utility have to be recalculated. The rollback algorithm can be fairly costly, with a worst-case time of O$(m * n^2)$ with n the number of resources and m unfulfilled plans. However in practice this algorithm won't be used all that often and even when it is used, the number of resources each plan requires is a very small subset of the total resources, so both the number of plans that need to be checked and the number of resources per plan that need checking are not all that large, making this affordable, even on a PDA.

```
for all resources used by the plan do {
   if the resource is not used by any other currently committed intentions do {
      unreserve it
      run the algorithm for a positive change in resources (3.3)
   }
}
```

Algorithm 3.4: Intention rollback

```
for all plans fulfilling the desire {
   if the plan is committed
      run the algorithm for the rollback of intentions (3.4)
   else
      set the plan's utility and formedness to 0
}
```

Algorithm 3.5: Desire rollback

Changes to other agents' plans Although we handled changes to other agents' plans as a separate change to contexts in 2.2.2, we already mentioned that this can be modeled through the communication with that agent. The communication with other agents will always be handled by a context-agent and can thus be seen as a change to relevant information.

Changes to desires

There are 3 possible changes to desires, which we will handle in 2 different ways. Firstly a desire may be fulfilled. Not as fulfilled according to the brain-agent, which is simply the fact that an intention has been committed to, but when that plan has been executed. In this case the brain-agent will be notified by the intention-agent about the fulfillment of the desire. This will be handled in the same way as the user manually deleting a desire (because he no longer desires it). The other situation is the addition of a new desire.

Desire removal In the case of a desire being fulfilled it depends on what kind of desire it is. A one-off desire will simply be deleted, but continuous and recurring desires are treated differently. In the case of a *user* deleting a desire the type of the desire is irrelevant and it will simply be deleted. In all of these cases we will use algorithm 3.5. If the algorithm was called due to fulfilling a one-off desire or deletion, we delete the desire, and all plans fulfilling it. In the case of a recursive desire we only remove the plans and desires, but activate a special metaplan that will add them again when the timer for the recurring interval is up. In the case of a continuous desire the formedness for the plans is set to 0 and it is treated as adding a new desire 3.3.2.

Adding a desire Adding a desire is pretty straightforward. Plans to fulfill it are requested from the intention-agent, these are evaluated in order of utility and if we can commit to one, we do so.

Modularity of actions

Because the communication between agents is asynchronous, we could have multiple triggers activated simultaneously. However, some actions need to be modular, so as to guarantee the stable state of the agent at all times. These actions are:

- Resource checking for a specific plan
- Reserving of resources

Both of these actions should lock any other actions from changing the resource- or planbase. However, the first operation may include communication with context-agents. We do not want our brain-agent locked while waiting for communication, so we will only apply this to local resource checking in the brain-agent's resourcebase. The way we solve this is by doing all checks on the local resourcebase and if a resource needs to be requested from a context-agent we send a request without waiting for an answer. The answer is then treated as any other change in resources.

3.3.3 Plans and the Intention-agent

Until now we have given a description of the context-agents and the brain-agent, but have largely ignored the intention-agent. This is because our research won't focus on the actual generation of plans. We assume some kind of plan generator, that generates plans to fulfill desires. The theory proposed here is independent of this plan generator, due to the use of the abstraction from plans in the brain-agent. However, to actually execute the plan decided on, we must always revert to the concrete plan and take the actions required. Thus the intention-agent's role is twofold: firstly it is the 'actor' in that only the intention-agent initiates actions on the system's behalf. Secondly it must 'generate' plans successfully and discover which resources are required to execute the plan. Assuming the plan generator and its subsidiary to distill the resources the intention-agent's role is rather straightforward: it will generate plans to fulfill desires, from which we will extract the necessary planning rules, which have sets of resources that need to be available to fulfill a desire. The intention-agent's role in our system is therefore to generate these planning rules from the more concrete plans as a layer of abstraction for us to work with.

3.3.4 Communication

As we have seen in the previous section, most actions in the brain-agent are started by communication from the context- or intention-agent. We therefore need to define messages that may do so. The messages we will handle in our multi-agent system are:

- Requesting a resource check (brain $\rightarrow$ context)
- Notifying of a resource change (context $\rightarrow$ brain)
- Requesting the plans for a desire (brain $\rightarrow$ intention)
- Answer with the plans for a desire (intention $\rightarrow$ brain)

- Notifying of the commitment to a plan (brain → intention)
- Notifying of the fulfillment of a desire (intention→brain)

And we see we will be communicating some of our datastructures. The syntax we will use for that is:

```
plan: plan(resource string, ..., resource string -> desire)
desire: desire(description)
resource: resource(description, parameters)
list: entry ; ... ; entry
```

3.4 Supporting human users

Ultimately the goal of the tool is to support a human user with impromptu coordination. It is therefore important to communicate important changes to him, but also to ask permission for certain actions. Most notably, to commit to a plan, the user must give his approval. If there are alternatives (more than 1 n-formed intentions for the same desire), he can choose, however in general the plan with the highest utility should be selected for his approval. Only after the user has given permission should an intention be committed to.

Usually execution of a plan by the intention-agent will require communication with the user. It may just be to inform him that something has automatically been done (ordering a book online), but most of the time the execution of a plan requires action from the user. Generally the user must go somewhere, do something or give the intention-agent permission to execute the plan. All of this communication will be encapsulated in the intention-agent's plan. Usually execution of a plan by the intention-agent will require communication with the user. It may just be to inform him that something has automatically been done (ordering a book online), but most of the time the execution of a plan requires action from the user. Generally the user must go somewhere, do something or give the intention-agent permission to execute the plan. All of this communication will be encapsulated in the intention-agent's plan.

However, more communication is necessary for the program to support the user fully. It must notify the user if an intention he was already committed to is rolled back, or he may wish to stay up to date on the state of the program. In case of failures, the user needs to be notified, but must also be told why the plan failed. To keep the user informed in this manner, each plan and each resource needs to be described in legible language, rather than the internal structure used. That way the user can see what resources he needs and what's available. He can see why intentions failed and the program takes on a true supportive role, rather than a black box which tells the user what to do.

Some of the aforementioned plans and actions are fairly invasive in the user's life. We therefore feel the buddy paradigm is almost a necessity for such a program to be successful. The program is helping the user to coordinate his life and a certain intimacy is required. The user must feel the program has his best intentions at heart, so that even though it is ‘merely min-maxing with abstract rules’, it seems as though it really cares. Although not always a good thing, letting the user anthropomorphize the agent to a certain extent can make him more comfortable [33]. In addition, the level of invasiveness of the program needs to be scalable. Some users may want to leave the program entirely free to schedule what it calculates as the best options, only being notified when they are required to act. Others may want to hold the system on a short leash and wish to know of every change that happens.

Our architecture leaves this communication completely open. The intention-agent has full knowledge of the available plans and the plans that are committed to, with all the resources required, so both ends of this scale are achievable. We leave it up to future design to actually create this buddy system on top of the underlying architecture.

3.5 A practical example

We will show how the situation detailed in the example of chapter 2 could occur within the framework outlined in this chapter. For this our system has the following context-agents:

- Location agent
- Online information agent
- Time agent

In addition the system obviously also has a brain- and an intention-agent. Here are the beliefbases of each agent:

Location agent Its beliefbase is a collection of GPS coordinates for interesting locations, including at all times that of the user. It contains the location of the bookstore as well, these two we will summarize as being:
me(1,1)
localstore(10,12)

Time agent The time context-agent is in charge of tracking the user’s appointments, deadlines and meetings. Its beliefbase is therefore a calendar. It also keeps track of the current time, so time constraints can be included in resource rules. We will write its beliefs as atomic sentences, disregarding how they are actually stored:
deadline(book, 28-11-2006)
appointment(28-11-2006)
now(24-11-2006 12:00:00)

Online information agent This agent doesn't really have a beliefbase as such, but rather has the capability to request information from middleware agents[24] and web services[13]. For our example we propose it connects to the websites of nile.com, stablesandbase.com and that of the local bookstore. It can request pricing information about books. Once requested, this information is kept as its 'local' beliefbase for future reference. This beliefbase will have the form of a relational database and after querying the relevant services will contain:
sale(nile, book, 30, instock, 5 days, online)
sale(localstore, book, 33, unknown, offline)
sale(stablesandbase, book, 32, instock, 3 days, online)

Brain-agent The resourcebase is the brain-agent's beliefbase, initially it is empty, however, resource rules can be used to generate the beliefs from the context-agents' beliefbases.

Intention-agent The intention-agent's beliefbase starts empty. It is only when desires are added and it is required to generate plans it will add these plans to its beliefbase.

3.5.1 Agent execution

We initialize the agent with just one single desire get(book). This desire is sent to the intention-agent, which uses its plan generator to form 3 separate plans. One to buy the book online from nile.com, one to buy the book online from stablesandbase.com and a third to buy the book from the local bookstore. It generates the plan rules based on the resources required and sends these back to the brain-agent:

- `plan1= plan(`"`info_available(nile, book, online), cost`",
  "$\forall$ `store2, item2: [info_available(store2, item2, method2), cost2]` $\wedge$ `cost` $<$ `cost2`",
  "`time_before_needed(book, T)`$\wedge$`info_deliverytime(nile, book, D)`$\wedge$`D` $<$ `T`"
  `-> get(book))`

- `plan2= plan(`"`info_available(stablesandbase, book, online), cost`",
  "$\forall$ `store2, item2: [info_available(store2, item2, method2), cost2]` $\wedge$ `cost` $<$ `cost2`",
  "`time_before_needed(book, T)`$\wedge$`info_deliverytime(stablesandbase, book, D)`$\wedge$`D` $<$ `T`"
  `-> get(book))`

- `plan3= plan(`"`info_available(localstore, book, offline), cost`",
  "$\forall$ `store2, item2: [info_available(store2, item2, method2), cost2]` $\wedge$ `cost` $<$ `cost2`",
  "`distance(localstore, X)`$\wedge$`X` $<$ `5`"
  `-> get(book)`

It then checks for the relevant resources, by querying the context-agents for the relevant resources. The information agent is queried about the info_available and info_deliverytime resources. The time agent is queried about the time_before_needed resources. And the location agent is queried about the distance resource.

These all add the exact resources requested into their list of relevant resources and use their resource rules (plans) to check the availability of all the requested resources). The resource rules are:

$$location(x,y) \wedge me(x2,y2)$$
$$\mapsto < distance(location, \sqrt{(x-x2)^2+(y-y2)^2}), \sqrt{(x-x2)^2+(y-y2)^2} >$$

$$sale(store, item, price, \texttt{instock}, deliverytime, \text{`}online')$$
$$\mapsto < info\_available(store, item, \texttt{online}), price + T * deliverytime >$$
$$sale(store, item, price, \texttt{instock}, deliverytime, \text{`}online')$$
$$\mapsto < info\_deliverytime(store, item, deliverytime), price + T * deliverytime >$$
$$sale(store, item, price, stock, \texttt{offline})$$
$$\mapsto < info\_available(store, item, \texttt{offline}), price >$$

$$deadline(item, date) \wedge now(now\_date\ now\_time)$$
$$\mapsto < time\_before\_needed(item, online) date - now\_date >$$

These result in the following resources updated in the beliefbase of the brain-agent:

Available:
info_available(nile, book, online), 45
info_available(stablesandbase, book, online), 41
distance(localstore, 14.2)
info_deliverytime(nile, book, 5)
info_deliverytime(stablesandbase, book, 3)
time_before_needed(book, 2)

Unknown:
info_available(localstore, book, offline), 33

All the abovementioned beliefs are marked as being relevant, because all of them are used in generating relevant resources.

And we see that we cannot commit to any of the 3 plans, because for each there is a resource check failing: the distance is greater than 10 and the delivery time for both nile and stablesandbase are too long. However, the user lives is a dynamic world and things change: he moves around town and the location agent's beliefs are updated: me(8,10)

This belief is relevant and thus we recalculate the resource, resulting in: distance(localstore, 2.8)

This means the resource checking is redone for plan3, and we see that it's still not known whether the book is available from the store. So that is queried again. The information agent recontacts the bookstore and finds out that they have sorted out their stock by now and know it is in stock. So <info_available(localstore, book, offline), 33> is now marked as being true, which means plan3 becomes 1-formed. This, coincidentally is also n-formed, because it only has 1 level of resource checking. We now ask the user whether he wants to buy the book from the local store and commit to this plan. If so, the intention-agent is given the go ahead and the plan is marked as committed.

Chapter 4

A JACK Prototype

We implemented the agents designed in 3 in the programming language JACK [25, 15]. JACK Intelligent Agents is a framework in Java for multi-agent system development. JACK uses a modular approach with which to build agents. The agents then communicate with each other using a portal. All communication is achieved in the form of Events. The special MessageEvents are events that are sent between agents, while the normal Event and the BDIEvent are used to trigger responses within an agent and control the flow of the program. A very useful addition for us is that events can be generated by changes in the beliefbase.

Events are handled by plans. The body of a plan contains regular java code, augmented with JACK-specific reasoning methods, which are compiled into java code at compile time. Behind the scenes a plan is compiled to a finite state machine, where each state is a statement in the plan. If any statement fails the plan fails.

Beliefs are declared separately and are simplified versions of relational databases. For more complex, beliefbase spanning datastructures, or java datastructures we wish to approach as a beliefbase, the View construct is used. The same queries can be defined as in Beliefs, but a view may encompass multiple beliefs and java datastructures to incorporate more complex methods into the JACK framework.

Capabilities are used as an encompassing collection of plans, events and belief types that fulfill a defined role within an agent.

JACK is one of the few Agent programming languages offering a runtime environment for PersonalJava, which runs on PDAs. 3APL-M[18] was another option we considered working with, however it works in J2ME, which was not available for the HP iPAQ at the time. In the end this wouldn't have mattered, because too many Java libraries are not available in PersonalJava to make it a really viable option. Our prototype system is not able to run on a PDA, because of the lack of these crucial libraries (e.g. the Collection framework). While these libraries are available in J2ME, there is no JACK runtime environment for this java version. So, due to these technical problems we chose to develop our prototype running on a desktop. The behaviour of a PDA can then be simulated.

4.1 Our implementation

We thought of including the project report generated by the JACK Development Environment, but it only describes the general structure of the system. Of far more significance is the way our deliberation cycle is implemented, as well as resourcebase construction and the other algorithms used. We will discuss them in detail here, however first we must admit that the way we describe the framework in chapter 3 fails to mention a few unsolved technical hurdles. We have to exclude or work around these in the implementation of the prototype. We leave these issues unresolved for now, for the sake of showing that the framework itself is practical and useable:

- The biggest is that we do not allow the use of predicate logic anywhere in the system, but most notably in planning rules. This is a fairly straightforward concession, but with large implications. It means that for a plan to succeed the resources must be defined precisely. You cannot have a plan rule stating $freetime(0.5hours)$, but must in fact use the exact times required, which is often not what is wanted. The solution would be to allow a first order predicate logic to be used in plan rules, with the inclusion of the java prolog engine, or by building a separate logical system in java. The inferring that has to be done is not extremely complicated and logical resolution should work fine, however incorporating this in the program is a lot of work.

- Another concession is that we did not implement any user interaction. The intention-agent has been stripped of virtually all possible functionality except the bare necessities. Its two main tasks should be to generate and execute plans, and to communicate these to the user. Both of these fall outside the main scope of our research and are not in our prototype.

- The algorithms implemented are neither the fastest nor the most efficient way of doing things. They could most probably be speeded up considerably with a fair amount of code optimization. There are many synchronization steps which could be skipped and also some of the algorithms cycle needlessly many times through all the plans, however this was done as a compromise between running speed and ease of programming. For the limited scope of the prototype, we do not need optimized code, so often we chose a simpler algorithm over a faster, but more complex one. We feel it is sufficient to show in our analysis that the algorithm can be optimized.

- The program in its current state cannot run on a PDA. We use parts of the Java libraries that aren't available for PersonalJava, which is what the JACK Runtime library runs on. J2ME does include these libraries, but isn't implemented for the HP iPAQ.

- The context-agents do not, as prescribed, use existing data structures for their beliefbases, such as a GPS system or a calendar, but instead use the JACK beliefbase. Also we don't use any middleware or use any of the announced internet services, due to practical issues with ontologies. There is still a lot of research being done into ontologies and, although very interesting, it is outside the scope of this project.

None of these are insurmountable. Each of them can be technically implemented, however this is primarily a research project and these are software engineering problems. We are confident our program demonstrates the feasibility of our theory and architecture. That is the goal of this prototype, however for a true implementation of the system, these engineering problems must be resolved.

4.2 An implemented MAS

As described throughout the document we have split our system up into several agents. Various context-agents, the brain-agent and the intention-agent. We will take a closer look at each of them, their implementation and the communication between them.

4.2.1 Beliefs

As described in section 3.2.1, the beliefs of the system are the beliefbases of the context-agents. We have implemented 3 context-agents. A location context-agent, a time context-agent and an information context-agent. The time and location contexts are real contexts which we described in section 1.1.1, but the information context is a catch all we use to be able to implement a couple of examples, like the one of buying a book, as in sections 2.4 and 3.5. Although normally the context-agents would use the underlying PDA's datastructures for the representation and be an extra layer on top of these, translating the information there into our own resources, in our implementation we chose to store the information in JACK beliefsets. The time context-agent keeps track of its own calendar and appointments are stored as 2 integers and a description. Whether time is available is simply calculated when asked for and if there isn't an appointment scheduled it is assumed to be available. Locations are stored very similarly, except that the 2 integers are of course coordinates and not begin and end times. A location we assume always to be known is our own location and distance is calculated simply as the length of the straight line between two points. The advantages to this approach are obvious: they are extremely simple and yet powerful enough to exhibit the potential application to real life examples of calendars and gps coordinates.

The information agent, however, is not very practical in a real life example, as we simply use it to store all other contexts. The availability of a book from a certain bookstore, or the availability of friends for an appointment are all stored here as atomic values, with no communication required. This is obviously a simplification of the more complex contexts which may also be dealt with, however for our prototype we do not really need them and chose not to extend our program in such a way.

4.2.2 Desires and Resources

Desires and Resources are modeled in the brain-agent. In JACK there is not a specific 'desire' datatype, so we model desires in a JACK beliefbase. However, because of the event driven nature of JACK, we can treat the desires as in normal BDI theory as the driving force behind the agent. Desires are the motivation for the brain-agent to do anything. So although the datatype they are stored in is a 'beliefbase' they are treated purely as desires. Changes to desires trigger deliberation directly and resources are marked as relevant if they have any effect on plans able to fulfill the agent's current desires. Resources are also modeled in a JACK beliefbase. A so-called open-world beliefbase. This is basically two beliefsets: one which evaluates to true and one which evaluates to false. Anything that is not in either beliefbase is treated as unknown. JACK is perfectly capable of treating with 'true' beliefs, however we are often forced to treat 'unknown' and 'false' beliefs in the same manner, because access methods return false in either case. We used a couple of work-a-rounds, however it is hard to use a 3-valued logic in JACK.

4.2.3 Plans and Intentions

Plans, remarkably enough are also stored as a beliefbase in the brain-agent, albeit a more complicated one. For plans we store the resource strings, its formedness, the desire it fulfills, its utility and whether it is committed to already or not. This may seem strange, because plans in the traditional sense are sequences of actions to be executed, but to us it is merely a couple of resources that need to be checked. The actual execution is done by the intention-agent. However, we conveniently ignored the plan generation in the intention-agent. The plans in the intention-agents are stored 2-fold. Firstly as a plan in the traditional sense and secondly as a belief, equal to the one in the brain-agent, but only with resource strings and the desire to be fulfilled. This is where the brain-agent gets its plans from after all. Thus the JACK plan constructs for the brain-agent do not in fact represent plans, but rather the deliberation cycle. We will discuss this further in section 4.3.

4.2.4 Resource rules

Unlike the brain-agent's plans, the context-agents' plans can be implemented as JACK plans. Each context-agent has plans to check for changes to relevant resources if its beliefs change. These plans contain the resource rules. Because it is very hard to use predicate logic in JACK plans, our resource rules are somewhat more restricted than we proposed in chapters 2 and 3. In some cases we can loop over the beliefbase, but in general this is not a viable alternative. However, for each context-agent we can implement the specific resource rules as we choose, so in some cases the model is more extensive. In most cases, however, the resource rules are simple boolean logic.

4.3 Deliberation

Now that we have explained how all the components of our framework are structured, we can focus on how they all come together.

All the more complicated calculations are taken care of in the brain-agent. It may send MessageEvents to the intention-agent asking for the plans available for new desires and to the various context-agents asking after the availability of certain resources. It then uses the algorithms described in 3.3.2 to recognize opportunities for impromptu coordination. We will explain the implementation of these algorithms in greater detail below. Because of the possibility for asynchronous evaluation of contexts, it has to react to updates from the context-agents often and restart in new parts of its deliberation cycle. These may overlap, but will never interfere with the constant condition we have in our system: **at all times we have a maximal set of n-formed plans, which is guaranteed to be maximal under the currently available resources**.

4.3.1 The algorithms

In section 3.3.2 we gave the general structure of the algorithms we need, but here we will look closer at the actual algorithms, most notably some of the more technical parts of algorithms that got glossed over in the previous chapter.

Resource checking

All of the algorithms described, must check the required resources of the plans at some time or another. The algorithm to do so is described in 4.1, which we will also use to describe a bit more of the JACK programming language. This is part of a reasoning method in a plan. Each statement of a plan, may be a java statement, a reasoning method statement or a logical expression. If a reasoning method statement or logical expression fails, and the failure isn't explicitly allowed for by putting it in the condition clause of an if or while statement, the plan fails. That is why the algorithm as described works. For each resource level after the n-th it tests whether the resources in that level are known to be available. If they are not in the resourcebase, then it sends a 'check'-event. This event is handled by another plan, which asks the relevant context-agent whether the resource is available. If this plan succeeds we still have to check whether the resource is really available. We update the i-formedness of the plan as we go along.

```
Vector levels = pr.getPlanResourceLevels();
Iterator it = levels.iterator();
int i_formed = n_formed;
while (it.hasNext()) {
  PlanResourceLevel level = (PlanResourceLevel) it.next();
  if(level.getLevel() == i_formed + 1) {
   Iterator itt = level.getResources().iterator();
   while(itt.hasNext()) {
     ResourceStruct r = (ResourceStruct) itt.next();
     String desc = r.name;
     Variables vars = r.variables;
     logical int cost;
     logical boolean reserved;
     @test(resources.get(desc, vars, cost, reserved), check.check(desc, vars));
     resources.get(desc, vars, cost, reserved) && !reserved.as_boolean();
   }
   i_formed++;
   if(i_formed > n_formed) {
     int utility = Utility.calculateUtility(pd.worth, pr);
     plans.change(id, pr, pd, utility, i_formed, false);
   }
  }
}
```

Algorithm 4.1: Checking resources for a plan

Maximal plans calculation

For the calculation of a set of plans that maximizes utility we use a fairly simple tree searching algorithm. First we have to create the tree, which is done with algorithm 4.2. A fairly straightforward algorithm, except for the addition of a check for supersedence or conflict. We use supersedence slightly differently than defined in 2.12. There it was always one plan superseding another. For the calculation of the tree, however we don't really care which plan supersedes which, just as long as they fulfill the same desire. The addition of these clauses causes a tree that is created from 'left to right', as we only try to add new plans to the existing tree and never backtrack and rebalance the tree. Searching through the tree is now extremely simple. The highest utility will always be in a leaf. So we just search through the leaves of the tree to find the group of plans with the highest utility and return it. Adding a plan is now done in O(n) worst case time. Calculating the utility also takes O(n) time if we keep track of the leaves in a separate collection. In our implementation we don't, and the worst case time is therefore O(n^2).

```
public void add(int planID, int planUtil,
        Collection supersedes, Collection conflicts) {
  if(isRoot()) {
    foreach child {
      child.addIterative(planID, planUtil, supersedes, conflicts);
    }
    children.add(new PlanNode(this, planID, planUtil));
  }
}
public void addIterative(int planID, int planUtil,
        Collection supersedes, Collection conflicts)
  if(supersedes.contains(this.id) || conflicts.contains(this.id)) {
    return;
  }
  this.leaf = false;
  foreach child {
    child.addIterative(planID, planUtil, supersedes, conflicts);
  }
  children.add(new PlanNode(this, planID, planUtil + utility));
}
```

Algorithm 4.2: Creating the plan tree

Initialize

The initialization algorithm 3.3.2 from Chapter 3 is the most complex algorithm in the system. It combines multiple instances of both selecting the maximal planset and checking their resources. Most of these can be chained sequentially after each other. In the worst case we might have to check all the resources(n) for all the plans(m), which leaves us with an O(nm) algorithm. However, as mentioned in 3.3.2 we use a leveled approach to resources for exactly this reason. Each level is checked separately and if resources are found missing we stop checking until we are notified they are available again. In general the set of plans that needs to be checked will also only be a small subset of the total available plans and so even the initialization, which we may expect to take longer due to it being a full-scale planning algorithm is done in acceptable time.

4.3.2 Context changes

The previous algorithms are all used in calculating which plans are executable and are involved with n-forming plans. These algorithms are integrated into the deliberation cycle in such a way that they are triggered by relevant changes in the context-agents' beliefs. Each context-agent has its own beliefs, which may change at any time. When they change, the context agent calculates whether this impacts the availability of any relevant resources and for each of these resources it sends an "Interrupt" message to the brain-agent. This can either be a positive interrupt, if the resource change is positive, or a negative interrupt if the resource change is negative. These interrupts are handled by the brain-agent as described in the algorithms 3.2 and 3.3.

4.3.3 Communication

In JACK communication is done with MessageEvents. These are handled exactly as normal events, except that they have a sender and a receiver. These events are used to send the information as described in 3.3.4. Although they may be a means of communication these events are also used in the normal sense: they trigger a reaction in the receiving agent. An event has a body in which java variables can be used, which forms the vehicle of the communication. We have the following MessageEvents:

`Query(variables)` is used to query the availability of a specific resource from the relevant context-agent. This triggers the context-agent to add the resource to its list of relevant resources and to check its availability at that time.

`Answer(description, variables)` answers a query with the availability of a resource.

`CommitMessage(plan)` ask the intention-agent to commit to a plan. The intention-agent will then ask the user for confirmation. If the intention is confirmed the intention-agent will execute whatever actions are in the plan.

`CommitResponse(committed)` answers whether the system will commit to the plan (boolean). This message is sent from the intention-agent to the brain-agent. The brain-agent will reserve the resources required for the plan when this message is received, so as to prevent any conflicts in plans using the same usable resources.

`UncommitMessage(plan)` tells the intention-agent a plan that was previously committed to can no longer succeed. This message is sent after a change to resources shows that a plan is no longer executable.

`UncommitResponse(uncommitted)` answers whether the system will uncommit the plan. If so the brain-agent will roll back the intention, freeing up the used resources. The desire the plan was intended to fulfill is once again unfulfilled.

`RequestPlans(desire)` requests the plans available to fulfill the given desire. These are added to the brain-agent's planbase.

`Interrupt(description, variables, availability)` notifies the brain-agent that some resource has changed and plans need to be recalculated. This can be positive or negative.

There's one notable omission from 3.3.4. We don't send a message when a plan has actually been executed and the desire fulfilled. This is unnecessary without the intention-agent doing anything to fulfill the desire and merely notifying the user. If we respond that we commit to the message, we might as well treat it as a done deal.

4.4 Human Computer Interaction

Once again, an omission in our implementation. Because this is a prototype to demonstrate the framework we haven't worried about how to communicate with the user and involve him with the decisions being made by the system. This is, however, an important part of the program if it is ever to be developed. It should run on a PDA and the currently clunky UI should be easy to use, but more importantly, the intention-agent will have to communicate any plans and changes to them to the user. It is a system to support impromptu coordination, not take it over entirely. Often the user will have to make choices and may choose differently than the system, impacting which plans can be used. However, for now we just use a simple interface in which we can add desires and beliefs and see what plans are formed.

4.5 An implemented example

While the most important algorithms have been explained, this doesn't yet give the full view of how the program works. To give a more complete description of how the program works, it is necessary to run it in practice. The following transcript is debug information printed while executing the same example as in 2.4 and 3.5:

```
LOCATION: Add belief: Me posted
LOCATION: Belief added: Me[10,10]
LOCATION: Add belief: Localstore posted
LOCATION: Belief added: Localstore[27,14]
TIME: Add belief: Presentation posted
TIME: Belief added: Presentation[Multi-agent Systems, 18-11-06 15:00, 18-11-06 15:00]
TIME: Add belief: Now posted
TIME: Belief added: Now[14-11-06 14:00]
INFORMATION: Add belief: information posted
INFORMATION: Adding information about: sale[Nile, Multi-agent Systems, 30, 5, online]
INFORMATION: Belief added: information(sale[Nile, Multi-agent Systems, 30, 5, online])
INFORMATION: Add belief: information posted
INFORMATION: Adding information about:
  sale[Stablesandbase, Multi-agent Systems, 32, 3, online]
INFORMATION: Belief added:
  information(sale[Stablesandbase, Multi-agent Systems, 32, 3, online])
INFORMATION: Add belief: information posted
```

```
INFORMATION: Adding information about: sale[Localstore, Multi-agent Systems, 33, offline]
INFORMATION:Belief added: information(sale[Localstore, Multi-agent Systems, 33, offline])

BRAIN: The desire buyBook will be added
BRAIN: Desire buyBook had been added
BRAIN: Desire added: buyBook[Mulit-agent Systems, 300]
```

These are the beliefs and desires we preprogram in the system as a starting situation.

```
=============================================
============= Starting runtime ==============
=============================================
BRAIN: Starting deliberation cycle
BRAIN: Initializing...
BRAIN: Handling: buyBook
BRAIN: Getting plans for: buyBook
BRAIN: adding plan: 0 = <sale[Stablesandbase, Multi-agent Systems, 5, online], 32> AND
  <deadline(Multi-agent Systems, 3), 9> => buyBook(Multi-agent Systems);
BRAIN: adding plan: 1 = <sale[Nile, Multi-agent Systems, 5, online], 30> AND
  <deadline(Multi-agent Systems, 5), 15> => buyBook(Multi-agent Systems);
BRAIN: adding plan: 2 = <sale[Localstore, Multi-agent Systems, offline], 33> AND
  <distance(Localstore, 8), 0> => buyBook(Multi-agent Systems);
```

The brain-agent has requested the plans for desire buyBook[Multi-agent Systems, 300] from the intention-agent and that returned 3 different plans. Plan 0 is to buy it online from Stablesandbase, plan 1 to buy it online from Nile and plan 2 to buy it at the local bookstore, Localstore.

```
BRAIN: Calculating maxUtil set
BRAIN: Got maxUtil set
BRAIN: 0, 259.
BRAIN: 1, 255.
BRAIN: 2, 267.
```

The execution of algorithm 4.2

```
BRAIN: Checking Resources for Plan: 2
BRAIN: Resource checking event has been posted
INFORMATION: Checking availability of: sale[Localstore, Multi-agent Systems, offline]
INFORMATION: Added new relevant information
INFORMATION: Checking beliefbase
INFORMATION: sale[Localstore, Multi-agent Systems, 33, 3, offline]
INFORMATION: true
INFORMATION: Answer message sent
BRAIN: Answer received
BRAIN: Adding new resource: sale[Localstore, Multi-agent Systems, offline], 33, true
```

```
BRAIN: Resource added
BRAIN: Resource checking event has been posted
LOCATION: Checking distance of: Localstore
LOCATION: Added new relevant location Localstore
LOCATION: distance[Localstore, 18]
LOCATION: Answer message sent
BRAIN: Answer received
BRAIN: Adding new resource: distance
BRAIN: distance[Localstore, 18], 18, true
BRAIN: Resource added
BRAIN: Plan 2 missing resource distance[Localstore, 8]
```

The resources for buying the plan from Localstore have been checked and failed, because the distance to the store is 18 and not 8, as was required. However, we did add some resources with updated costs, so we need to recalculate the utilities of all the plans:

```
BRAIN: Recalculate utilities
BRAIN: New utility for plan 0: 259
BRAIN: New utility for plan 1: 255
BRAIN: New utility for plan 2: 267
BRAIN: Checking Resources for Plan: 0
BRAIN: Resource checking event has been posted
```

Due to the unavailability of plan 2, plan 0 now has the highest utility, so we start checking the resources for plan 0 in the same way as we did above for plan 2.

```
INFORMATION: Checking availability of:
  sale[Stablesandbase, Multi-agent Systems, 5, online]
INFORMATION: Added new relevant information
INFORMATION: Checking beliefbase
INFORMATION: sale[Stablesandbase, Multi-agent Systems, 32, 3, online]
INFORMATION: true
INFORMATION: Answer message sent
BRAIN: Answer received
BRAIN: Adding new resource: sale[Stablesandbase, Multi-agent Systems, 5, online], 32, true
BRAIN: Resource added
BRAIN: Resource checking event has been posted
TIME: Checking time resource: deadline
TIME: Added new relevant time resource
TIME: Now = Tue Nov 14 14:00:00 CET 2006
TIME: TimeNeeded = Sat Nov 18 15:00:00 CET 2006
TIME: Time till we need the item: 4
TIME: Answer message sent
BRAIN: Answer received
BRAIN: Adding new resource: time
```

```
BRAIN: time[deadline, Multi-agent Systems, 3], 9, true
BRAIN: Resource added
```

This time all the resources are available, however we need to recalculate the utilities of all the plans and make sure we don't now have a plan that could potentially give a higher utility.

```
BRAIN: Recalculating utilities
BRAIN: New utility for plan 0: 255
BRAIN: New utility for plan 1: 259
BRAIN: New utility for plan 2: 267
```

This is not the case, so we can commit to plan 1.

```
BRAIN: Resources available for plan with ID: 1, sending commit request
BRAIN: Starting commit procedure
BRAIN: Sending message to intention-agent
INTENTION AGENT: Handling commit of plan
INTENTION AGENT: Do you wish to commit?
n
INTENTION AGENT: Reply received, continuing
BRAIN: Not committing to plan: 1
```

However, the user was asked for input and didn't wish to, so we don't. The user goes on his business and at some time during the day his location changes:

```
LOCATION: Belief changed: Location, Me
LOCATION: ++++++++++++Some belief was added, reacting
LOCATION: Posted checking: Me[20, 13]
LOCATION: ++++++++++++Checking relevance
```

The user's position has changed, we need to check whether any relevant distances have changed. Due to the fact that it is the user's own position that changed we have to iterate through all the relevant positions and check them. In our case there is of course, only one, which happens to be relevant, because we are now within 8 units of the bookstore:

```
LOCATION: Handling my own position change
LOCATION: Sending interrupt
BRAIN: ++++++++++++INTERRUPT
BRAIN: Handling positive interrupt
BRAIN: Resource changed
BRAIN: Changed resource, now handle deliberation
BRAIN: distance, [Localstore, 8]
BRAIN: Plans requiring said resource: [2]
BRAIN: Reforming  plan: 2
BRAIN: Checking Resources for Plan: 2
BRAIN: sale[Localstore, Multi-agent Systems, offline], 33, true
BRAIN: distance[Localstore, 8], 0, true
BRAIN: All resources evaluated to true
```

An interrupt event is sent to the brain-agent, notifying it of the resource change and giving it the updated resource. This is added to the resourcebase, after which all the plans requiring this resource are checked again. In this case just plan 2, which now has all resources available to it. So we will attempt to commit plan 2, because we refused to commit plan 1 and thus the desire `buybook[Multi-agent Systems]` is still unfulfilled.

```
BRAIN: Resources available for plan with ID: 2, sending commit request
BRAIN: Starting commit procedure
BRAIN: Sending message to intention-agent
INTENTION AGENT: Handling commit of plan
INTENTION AGENT: Do you wish to commit?
y
INTENTION AGENT: Reply received, continuing
BRAIN: Committing to plan: 2
BRAIN: Resources locked
BRAIN: Committed to plan: 2
BRAIN: buyBook, [Multi-agent Systems]
BRAIN: The desire buyBook will be fulfilled
BRAIN: Desire buyBook had been fulfilled
BRAIN: Interrupt done
```

This time we agreed to commit. The intention-agent does nothing, however in a real life situation it could reserve the book at the bookstore and show the user a route how to get to the bookstore. The brain-agent marks the belief as fulfilled and reserves all the resources required for the plan. The plan is marked as committed.

Chapter 5

Conclusions and Future Work

5.1 Experiences with JACK

Although Agent Technology is a rapidly growing field, and situated awareness is becoming a more and more predominant subject, it is still very hard to find an agent programming language suited for the task. We picked JACK because it can run on PDAs and its event-driven approach seemed ideally suited to the kind of application we were developing. However, even a commercial product like JACK is not without its flaws. There are numerous bugs, especially in the previous versions we used and not everything could be implemented the way we wished. JACK is missing a real logic interpreter and as such plan rules and resource rules are sorely lacking in the current implementation. There is supposed to be support for an ‘open world’ beliefbase, where we can use three-valued logic, but in practice it is impossible to evaluate any such three-valued logic, as `unknown` is equal to verb|false| for all intents and purposes.

Even so, all in all, JACK is probably the best choice for many agent applications. It is based on java and in almost every part of the language you can insert java methods, letting you extend the framework with custom datatypes and algorithms. The reasoning methods are a powerful tool for evaluating success of a plan, but there is a steep learning curve in managing the intricacies of the programming language. The documentation seems extensive, but is sorely missing examples and explanations of the more complicated concepts of the language, such as belief callbacks and the use of cursors to compare specific beliefs. These concepts are implemented in the language, however it is necessary to read through the generated java code to figure out how they work exactly, because even the API doesn’t give a full explanation of many concepts.

Learning how the programming language works and finding and working around bugs took up a lot of time. The error messages are often vague (or non-existent) and the actual error often occurs in an entirely other part of the program. In this it differs a lot from java, in which the errors are clear and always pointing to the exact spot in the program where it derailed, both while compiling as when running the program.

When the programming language was 'under control' it was pleasant to work with and we feel that, in some situations, it compares favourably to other agent programming languages, such as for instance 3APL [8, 14]. While it misses a usable logic engine, its interface with native Java code is more natural. However, this thesis is not about a comparison between various programming languages, but about the specific application we developed. In our case JACK was an expedient choice, which, with numerous bumps in the road, worked out well.

5.2 Conclusions

In chapter 1 we described the characteristics of impromptu coordination and the problems we encounter when writing software to assist with it. The central question we asked was:

> **How can an agent recognize and help react to opportunities for fulfilling desires in changing contexts?**

An agent can recognize opportunities by being aware of the various contexts it resides in and realizing when changes in these contexts are relevant for it. Relevance is defined by which resources are necessary for it to fulfill its desires. Relevant changes in context can be positive, in which case we may have an opportunity to fulfill more desires, or else they can be negative and we may have to roll back plans and can no longer fulfill certain desires. In either case the agent can recognize them. The plans an agent has are executed based on the resources available. New resources becoming available triggers new plans to executed. These help the user react to opportunities or setbacks.

Chapter 2 outlines a logical structure which gives a clear definition of the process of reacting to new information. We give a working definition of what impromptu coordination is through the definition of 'opportunity' 2.14 and outline the logical structure of an agent to deal with it. In chapter 3 we describe an architectural framework for a multi-agent system, comprised of several BDI agents to deal with this problem. We believe this framework is effective at dealing with the problem and the first results of running the prototype on some examples support us. It is eminently capable of dealing with the example in section 2.4, but also with some of the other examples scattered throughout chapter 1. We did encounter certain limitations, which is why the framework in 3 is more restricted than the logical outline in 2. We enforce more restrictive logics and omit complex issues such as ontologies to communicate with other agents and plan generators to deal with new desires. In chapter 4 we narrow the scope down further with the implementation of a JACK prototype. The limitations of a concrete programming language enforce further restrictions. In JACK this was the limited use of predicate logic, but other languages have different restrictions. JACK left us free to implement the deliberation cycle largely how we saw fit, which worked out very well, because that is ultimately what we wish to demonstrate. We have analyzed the more complex algorithms used in the system and verified that the running time for them is acceptable for use on a PDA, as long as the underlying datastructures are implemented efficiently. Each of the algorithms has a worst case complexity of O(nm), but even the initialization algorithm which necessarily must calculate the most has sufficient optimizations built in that the real complexity will be a mere fraction of that. The main problem would normally be the sheer amount of information that a context aware application needs to work with, but our preprocessing in the context-agents takes care of this, by verifying the relevancy of the information. In addition these agents are not bound to run on the mobile device. Due to the distributed nature of the system, they may run on a server and use the larger resources available there for the preprocessing if it's necessary, further freeing up the limited resources of the mobile device.

5.3 Future work

We have already highlighted the main points for future research in the rest of the document, but to really create an agent system supporting impromptu coordination there is still a lot of work to be done.

- To truly react to opportunities, the plans must be more sophisticated. The prototype implementation of the intention-agent only contains the bare necessities for functioning. The execution of a plan is yet unimplemented. Also, the plans it has need to be implemented by a human. This planbase needs to be extended, or replaced entirely with some form of plan generator to come up with plans for some of the desires of a user. This plan generator must also distill the resources required for the plan's execution and create the abstract plan we deal with in the brain-agent.

- To support a human user there must be a clear method of communication between him and the program. Because the program is a system of largely autonomous agents, the user needs to be able to convey information about himself and the world to the context-agents. More importantly, however, the intention-agent must be able to convey new plans, changed plans and possible plans in a dynamic manner. Conventional user interfaces are not concise enough for use in short times on a small mobile device. A new user interface is needed, that helps deal with the complexity of the problem. Using the buddy paradigm such a UI should be possible of narrowing the gap between the agent and the user.

- To enhance the context-awareness of the program and recognize changes in the contexts we described in 1.1.1 we will need more enhanced context-agents. We implemented 3 different context-agents, but they merely simulate true context changes through artificially added beliefs. The location agent is not connected to a GPS receiver, nor is the time agent connected to a calendar. To seriously consider an agent context-aware it needs more than that:

  - Full fledged context-agents, using the available data on the PDA and other sources instead of a redundant internal structure.
  - Ontology support to communicate with other agents, most notably on internet.

  Especially the second item is far from trivial. While a limited amount of ontologies could be preprogrammed, there are too many different systems to preprogram them all. Automatic ontology translation is an active research area, with many unsolved issues.

- To be able to reason about beliefs in resource rules and about resources in plan rules, a prerequisite is some form of predicate logic. I already mentioned this notable absence in JACK, so the system must either be programmed in another language, or a logic interpreter must be plugged into JACK. Either way there are still all the problems associated with logic programming.

- And finally the system must actually run on mobile devices. The system is specifically designed with a mobile device in mind. It is a great pity the prototype cannot be demonstrated in its intended environment. For it to be of any use, a next implementation should have the brain-agent and some context-agents working on a PDA and other context-agents connecting over internet.

Designing and developing these to fit within our framework would give a wonderful overview of the system as a whole, but if designed well would probably even serve as a commercial product!

Bibliography

[1] N. Alechina, M. Jago, and B. Logan. Resource-bounded belief revision and contraction. In *Proceedings of the 3rd International Workshop on Declarative Agent Languages and Technologies (DALT 2005)*, July 2005.

[2] L. Barkhuus and A. Dey. Is context-aware computing taking control away from the user? three levels of interactivity examined. In *Proceedings of the Fifth Annual Conference on Ubiquitous Computing (UBICOMP 2003)*, pages 149–156, 2003.

[3] R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23, 1986.

[4] H. Chalupsky, Y. Gil, C. A. Knoblock, K. Lerman, J. Oh, D. V. Pynadath, T. A. Russ, and M. Tambe. Electric elves: Agent technology for supporting human organizations. *AI Magazine*, 23(2):11–24, 2002.

[5] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[6] K. Cheverst, N. Davies, K. Mitchell, and A. Friday. Experiences of developing and deploying a context-aware tourist guide: the GUIDE project. In *Mobile Computing and Networking*, pages 20–31, 2000.

[7] P. Cohen and H. Levesque. Intention is choice with commitment. In *Artificial Intelligence*, volume 42, pages 213 – 261, Essex, UK, 1990. Elsevier Science Publishers Ltd.

[8] M. Dastani, B. F., D. F., H. W. van der, K. M., and M. J.J. Programming the deliberation cycle of cognitive robots. In *Proceedings of the The Third International Cognitive Robotics Workshop*, 2002.

[9] M. de Weerdt, A. Bos, J. F. M. Tonino, and C. Witteveen. A resource logic for multi-agent plan merging. *Annals of Mathematics and Artificial Intelligence, special issue on Computational Logic and Multi-Agent Systems*, 37: 93–130, 2003.

[10] P. Gardenfors. *Belief Revision.* Cambridge University Press, 2003.

[11] C. Graham and K. Cheverst. Guides, locals, chaperones, buddies and captains: managing trust through interaction paradigms, 2004. URL http://www.mguides.info.

[12] W. G. Griswold, R. Boyer, S. W. Brown, T. M. Truong, E. Bhasker, G. R. Jay, and R. B. Shapiro. Using mobile technology to create opportunitistic interactions on a university campus. Technical report cs2002-0724, Computer Science and Engineering, UC San Diego, September 2002.

[13] J. Hendler. Agents and the semantic web. *IEEE intelligent systems and their applications*, 16(2):30–37, 2001.

[14] K. V. Hindriks, F. S. d. Boer, W. v. d. Hoek, and J.-J. C. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2: 357–401, 1999.

[15] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. Jack intelligent agents – summary of an agent infrastructure. In *5th International Conference on Autonomous Agents*, 2001.

[16] A. K. Jessica Smith, William Mackaness and I. Williamson. Spatial data infrastructure requirements for mobile location based journey planning. *Transactions in GIS*, 8(1):23–44, 2004.

[17] M. Kakihara and C. Sørensen. Mobility: An extended perspective. In *Proceedings of the Hawaii International Conference on System Sciences*, January 2002.

[18] F. Koch, I. Rahwan, F. Dignum, and J.-C. Meyer. Programming deliberative agents for mobile services: the 3apl-m platform. In *Proceedings of the third international workshop on Programming Multi-Agent Systems*, pages 179–192, 2005.

[19] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[20] J.-J. C. Meyer and W. v. d. Hoek. *Epistemic Logic for AI and Computer Science*. Cambridge University Press, 1995.

[21] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. R. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3): 36–56, 1991.

[22] M. O'Grady and G. O'Hare. Mobile devices and intelligent agents — towards a new generation of applications and services. *Intelligent Embedded Agents*, 171:335–353, May 2005.

[23] K. O'Hara, M. Perry, and S. Lewis. Social coordination around a situated display appliance. In *Proceedings of the conference on Human factors in computing systems*, pages 65–72, 2003.

[24] B. J. Overeinder and F. M. T. Brazier. Scalable middleware environment for agent-based internet applications. *Lecture notes in computer science*, (3732):675–679, 2005.

[25] A. H. Paolo Busetta, Ralph Rönnquist and A. Lucas. Jack intelligent agents - components for intelligent agents in java. *AgentLink News*, 2, 1999.

[26] N. W. Paton and O. D&#237;az. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/311531.311623.

[27] I. Rahwan. *Interest-based Negotiation in Multi-Agent Systems*. PhD thesis, University of Melbourne, Melbourne, Australia, 2004.

[28] I. Rahwan, G. Connor, and L. Sonenberg. Supporting impromptu coordination using automated negotiation. In M. Barley, editor, *Proceedings of the 7th Pacific Rim International Workshop on Multi-Agents (PRIMA2004)*, Berlin, Germany, 2005. Springer-Verlag.

[29] I. Rahwan, F. Koch, C. Graham, K. J. A., and S. L. Goal-directed automated negotiation for supporting mobile user coordination. *Lecture Notes in Artificial Intelligence*, 3554, 2005.

[30] T. Rahwan, T. Rahwan, I. Rahwan, and R. Ashri. Agent-based support for mobile users using agentspeak(l). *Lecture Notes in Artificial Intelligence*, 3030:45–60, 2004.

[31] R. Ramakrishnan and J. Gehrke. *Database Management Systems*, pages 523–537. McGraw-Hill, 2 edition, 2000.

[32] 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. ISBN 1-55860-165-1.

[33] T. Riley. The anthropomorphizing of intelligent agents. In *FAABS '00: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers*, pages 323–334, London, UK, 2001. Springer-Verlag. ISBN 3-540-42716-3.

[34] R. Smith. The contract net: a formalism for the control of distributed problem solving. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, 1977.

[35] G. Wagner. Artificial agents and logic programming. In *panel statement at the ICLP'97 post-conference workshop Logic Programming and Multiagent Systems*, 1997.

[36] M. S. Winslett. *Updating Logical Databases.* Cambridge University Press, 1990.

[37] M. Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley and Sons, Chichester, England, 2002.

[38] M. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.