Bachelorthesis

Lucas Jenß

Automated meeting scheduling using multi-agent technology

Lucas Jenß

Automated meeting scheduling using multi-agent technology

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Kai von Luck
Zweitgutachter : Prof. Dr. Gunter Klemke

Erstellt am 27.12.2012

Lucas Jenß

Thema der Bachelorthesis

Automatisiertes Meeting-Scheduling unter Verwendung von Multi-Agent Technologie

Stichworte

Multi-Agenten, Meeting Scheduling, Companion-Technologie, Büroautomatisierung

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit dem Design eines verteilten Multi-Agenten-Systems, welches in der Lage ist, eigenständig Meetings mehrere Teilnehmer zu planen. Zu diesem Zweck wird zuerst der aktuelle Stand der Forschung untersucht. Anschließend wird der Vorgang des Planens von Meetings analysiert um die Anforderungen zu extrahieren, auf welchen das System basieren soll. Nachfolgend wird eine Architektur für ein solches System, sowie ein Kommunikationsprotokoll für die Agenten, erarbeitet, mit dem Fokus auf Flexibilität der Modifikation des Agentenverhaltens. Schlussendlich wird eine "proof-of-concept" Implementation vorgestellt, welche Anhand von Beispielszenarien evaluiert wird.

Lucas Jenß

Title of the thesis

Automated meeting scheduling using multi-agent technology

Keywords

multi-agent, meeting scheduling, companion technology, office automation

Abstract

This thesis deals with the complexities of designing a distributed multi-agent system capable of autonomously scheduling meetings for multiple participants. To this end, the current state of research is first examined. Subsequently, the problem of meeting scheduling is analyzed in order to extract the requirements on which the scheduling system shall be based. After that, an architecture for such a system is devised, along with a communication protocol for the scheduling agents, focussing on the flexibility to easily modify the agent's behavior. Finally, a proof of concept implementation of the architecture is presented and evaluated in example scheduling scenarios.

Contents

1 Introduction

The notion of computer systems that assist the user in tedious everyday tasks grows more important every day in today's fast moving domestic and professional life, and its importance is well reflected in the computer science research community. Yorik Wilks, an important contributor in the field, has extensively researched the key social, psychological, ethical and design issues that arise when admitting "Artificial Companions" into our society. Artifical companion systems are cognitive technical systems, which fully adapt their functionality to the abilities, preferences and requirements of their user. As such, a companion system should be capable of autonomously providing a reliable service, cooperating with the user where necessary. In 2009 the University of Ulm, the Otto-von-Guericke University in Magdeburg as well as the Leibniz-Institute for Neurobiology, inspired by the work of Wilks, founded the special research field "transregio 62", which focuses on the field of companion technology and has resulted in more than 350 publication to date. (*SFB Transregio 62*, 2012).

When speaking of tedious everyday responsibilities, one might reach out to the task of meeting scheduling as an example. For every group effort that is to be performed, meetings need to be held in order to divide effort into smaller assignments, distribute them across group members and later to synchronize the group's progress. Complex tasks or deep organizational hierarchies might even require several division steps until the individual can begin its work. From these observations as well as case studies covering the subject (e.g. Romano and Nunamaker (2001)), one can deduce that meetings are very commonly held in today's working environments, and thus many meetings need to be scheduled.

Meeting scheduling has received a large amount of attention from researchers in the past, and most of the systems that have been developed are based on the idea of autonomous entities that, by collaborating with each other, are capable of solving certain tasks. These entities are often called agents and their concept is very similar to the one of companion systems. Systems of multiple agents are often used for this kind of negotiation problem, because the abstraction of an agent representing a person's calendar and preferences is very close to the way humans communicate and negotiate when scheduling a meeting.

In the course of this thesis, a multi-agent system capable of autonomously scheduling meetings will be devised, based on the requirements that will be gathered while analyzing the task of meeting scheduling. The focus will lie on the system's design, with the goal of providing a flexible foundation architecture which future research can be based upon.

Outline

In Chapter 2 an overview of existing research on the topic of multi-agent meeting scheduling is given, positioning this thesis within the context of previous work.

Chapter 3 then analyzes the task of meeting scheduling based on a scenario extracted from previous research. The scenario analysis is performed to better understand the complexities of meeting scheduling and thus, in Section 3.3, be able to extract the requirements for a computer system capable of automatically scheduling meetings, which are then summarized in Section 3.5.

On the basis of these requirements, a multi-agent meeting scheduling system is designed in Chapter 4. In this chapter, the agent communication layer serving as the foundation for the system is described in Section 4.1, followed by an explication of the agents that the communication layer accommodates (Section 4.2). The chapter is concluded by an explanation of the devised meeting scheduling protocol used by the agents (Section 4.3), as well as an exemplary technique on the evaluation of multiple meeting time proposals (i.e. which meeting time proposal is best suited for all participants) in Section 4.4.

The multi-agent system designed in Chapter 4 is implemented as a prototype in Chapter 5, giving a short overview of the implementation environment (implementation language, employed frameworks, etc. [Section 5.1]) and the development approach (Section 5.2). Subsequently, the prototype implementation will be evaluated by means of the requirements gathered in Chapter 3, focusing on the confirmation of the design's hypothetical flexibility established in the previous chapter.

A summary of the conducted work and its results is presented in Chapter 6, along with future prospects and ideas on how the devised system architecture might be enhanced to overcome current limitations and make it easier adaptable to future requirements.

2 Literature Review

In his book “Society of Mind”, Minsky (1988) uses the term “agent” not to refer to a certain component of a computer system, but to a more abstract notion of a “mindless” process (when observed individually) that, when interconnected with many other such processes, forms an intelligent entity or “a mind” which is able to perform complex tasks. Consequently, in a computer system context, a (Multi-) Agent System can be seen “as a loosely coupled network of problem solvers agents (section 4.2) that work together to solve problems that are beyond the individual capabilities or knowledge of each problem solver”. (Jennings et al., 1998)

Meeting scheduling, that is, the activity of finding an appropiate time for a meeting, often with many participants, is a problem that requires the collaboration of many different entities in order to be successful. As such, its automation is a good example of how the inherent complexity of a problem can be distributed accross several problem solvers, thus simplifying the task each problem solver has to solve. The research field of automated meeting scheduling is large and many contributions have been made over the years. According to a survey by Kincaid et al. (1985), basic electronic calendars became widely available in business environments sometime between 1982 and 1985. Said survey also indicates that even then, the inclusion of an automated meeting scheduling system was a feature that users desired in the context of an electronic calendar system. This is unsurprising, given the fact that managers and executive staff spent close to 50% of their time in meetings (Teger, 1983; Mintzberg, 1973), and those meetings need to be scheduled, which is a very time consuming task when done manually (i.e. face-to-face, by telephone or by email), and the complexity of finding a mutually satisfactory time rises with the number of participants (Crawford and Veloso, 2005b).

Early automated scheduling systems include the Amoeba Diary System introduced by Johansen and Anshus (1988), which is not fully distributed however, “since it contains a so-called ‘global-module’ which acts as a centralized scheduling manager” (Mattern and Sturm, 1989). Another early representative is the Eden Shared Calendar System presented by Holman and Almes (1985) which is part of The Eden Project, an experiment in designing, building and using distributed computing systems. Because of the experimental nature of the project, the developed calendar system was mainly intended to evaluate the Eden system’s hospitability though

(Black, 1986). Mattern and Sturm (1989) are arguably the first to implement a fully distributed appointment scheduling system based on the notion of agents communicating solely through message passing, where every agent in the system belongs to a specific user and manages its user's private calendar and preferences. The developed prototype is said to be able to automatically schedule and re-schedule meetings using several different heuristics, none of which, unfortunately, are explained in detail. However, the mentioned heuristics are all based on the user's preferences, underlining the importance of knowing them in order to achieve a satisfactory scheduling result. Since then, the importance of knowing and improving the understanding of user preferences has been acknowledged by many sources (Oh and Smith, 2004; Crawford and Veloso, 2005a,b; Chun et al., 2003), which exclusively focus on the best way of learning and using the user's preferences to achieve optimal scheduling results.

As previously mentioned, early sources on the subject of meeting scheduling focus more on the distributed nature of the problem, paying almost no attention to the heuristics and algorithms involved in the scheduling process. Later works focus more on these formerly overlooked areas: Bui et al. (1995) formalize the problem of meeting scheduling, presenting an incremental negotiation scheme and process in the context of an agent-oriented application. In their approach, attributes are grouped hierarchically and are then negotiated in a top-down fashion. For example the part of a week might be scheduled first (early or late), then the exact day, morning or afternoon and finally the exact meeting time. Chun et al. (2003) take a different approach denoted "relaxation", in which the agents gradually "relax" their preferences (therefore widening the search) until a mutually acceptable solution is found.

However, few of the aforementioned sources give any insight into their meeting scheduler's system design, only exposing the predominant system features without giving an architectural overview of what has been designed and implemented.

In this thesis, a slightly different approach is taken. Based on the selection and combination of existing techniques, an overall multi-agent meeting scheduling system design is developed, focusing on flexibility to accomodate different scheduling techniques such as heuristics and preference estimation methods.

3 Analysis

During the course of this thesis, a multi-agent system capable of automated meeting scheduling will be designed and subsequently evaluated by means of a prototype implementation. The system's design will have a strong focus on behavioral flexibility, meaning that core modules implementing the agents' behaviors will be independently replaceable, thus resulting in a design that is both apt for further research and implementations destined for the end-user. As a multi-agent system, it will be capable of carrying out its functionality in a distributed manner without the requirement of any centralized entity. For such a system to be devised, the common approach to meeting scheduling, as it is carried out without the aid of automated systems, must be analyzed, with the objective of gathering the main requirements for successful automated meeting scheduling.

The presented meeting scheduling approach on which this analysis is based, is an aggregation of many published works on the subject of meeting scheduling[1], and as such it is considered the common-sense meeting scheduling scenario for the purpose of this thesis on which all further effort is based.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in all following sections of this thesis are to be interpreted as described in RFC 2119. (Bradner, 1997)

[1]The difficulty in citing concrete sources for the aggregation lies in the fact that much of the information presented is only implicitly contained in the published works, and much of it has been conceived by cross-referencing works of different authors in the course of the creation of this thesis. Influential works include Crawford and Veloso (2005c); Chun et al. (2003); Bui et al. (1995). It should also be noted that every source that has influenced the following analysis in any way is cited in the bibliography section of this thesis.

3.1 Meeting scheduling

When a meeting needs to be held, commonly a single person is assigned to invite all other people who are to come to the meeting (meeting invitees) and, in the end, to choose a time for the meeting to be held. This person is called the “meeting initiator”, who usually is a meeting invitee himself, or acts on behalf of one (e.g. a personal assistant or a team leader).

The meeting initiator is assumed to be aware of the constraints that have to be satisfied for the meeting to be successful, especially personal (i.e. which invitees *have* to attend the meeting) and time contraints (e.g. when is the latest possible time for the meeting to be held or when is the latest possible time at which a decision must have been reached). The difficulty for the meeting initiator now lies in the selection of a proper time for the meeting, so that at least all required invitees are able to attend, and at best all invitees are satisified with the time that’s been decided.

If the meeting to be planned lies sufficiently far in the future or if its importance is high enough so that the meeting initiator can assume that the invitees will reschedule meetings which collide with the one being scheduled, he can simply choose a fixed time for the meeting and communicate it to all invitees. This technique is commonly used for large meetings, e.g. talks and conferences. Given that there is no distributed effort involved in finding a feasible time using this approach, it is not further discussed here.

In this thesis, the typical approach for meetings that involve the need for a consensus on the decided meeting time is referred to as the communication centered approach, given that the meeting initiator “manually” communicates with all invitees in order to find a suitable meeting time.

3.2 Communication centered meeting scheduling

In the communication centered meeting scheduling approach, the initiator first chooses an initial time proposal, based on the constraints of the meeting and his own preference, of which he believes that it has a high probability of being accepted by the meeting invitees. He also checks whether or not all invitees actually want to attend the meeting in question, and discards those from the scheduling process who do not. All invitees that have explicitly stated their interest in attending a given meeting are regarded as “meeting participants”.

The initiator now inquires if the proposed time satisfies the participants. If it does not, the initiator may ask for any number of counter-proposals in order to select a new proposal. If all participants are satisfied, the meeting scheduling process was successful. If one or more participants rejected the proposal, a new one must be chosen, based on the information that the meeting initiator has acquired so far. It is important to note that very often the response of a participant regarding its satisfaction for a proposal won’t be *“I am free”* or *“I am not free”*, but something more vague, for example: *“I could make it if need be, but I would really prefer another time”*. Such a response is very difficult to represent in a formal manner due to the fact that the rationale behind the statement is unbeknownst to the initiator. Thusly, if several such responses are given by different participants, comparison between them poses a problem.

After a viable time for the meeting has been found and all participants have been notified, it is possible that the circumstances surrounding the meeting change, making it necessary to re-schedule or, if no alternative time can be found, cancel the meeting entirely. Re-scheduling a meeting is no different from the original scheduling process, except for the consideration of the information that has already been collected in the previous scheduling iteration.

The communication centered approach to meeting scheduling often requires several rounds of negotiation with all participants in order to reach a consensus (Chun et al., 2003), and as such can be very time consuming when done manually.

3.3 Requirements analysis

In this section the previously described meeting scheduling approach is analyzed, extracting from it the requirements for a software system capable of automatically scheduling meetings.

3.3.1 Stakeholders

Meeting initiator The meeting initiator is the person in charge of scheduling a meeting. He has to be able to start a new meeting scheduling process, supplying the information available to him (see “3.3.6 Information necessary for scheduling”). He must be able to rely on the correct functionality of the system. Otherwise, the resulting meeting times would have to be checked with every participant, which would lessen the system’s benefit. The initiator may also be a meeting participant.

Meeting invitee A meeting invitee is a person invited to a meeting who has not yet stated if he wants to follow that invitation or not. At the very least, it must be possible for him to state whether or not he wants to attend the meeting.

Meeting participant After stating the intention to attend a meeting, an invitee is considered to be a meeting participant. Every meeting participant’s interests must be represented by the system in order to achieve a fair meeting scheduling result (see also “3.3.6 Scheduling fairness”). The participants rely on the system’s impartiality, except for the participants’ priorities set by the meeting initiator. They also rely on the system’s correctness, especially in the sense that the system must never falsely assume a positive response for any proposal, because that would lead to meetings being scheduled at unfeasible times.

3.3.2 Terminology

Meeting proposal A meeting proposal is sent to an invitee asking him to attend a meeting at a proposed meeting time. It also has to contain additional information about the meeting, such as “location” or “topic”, so that the recipient may decide if he wants to attend the meeting or not.

Meeting counter-proposal The answer to a meeting proposal, in case the recipient of the latter is not able to attend the meeting at the proposed time. It must contain an alternate date and time at which the sender would like the meeting to be held. It may contain additional proposals such as an alternate location for the meeting.

3.3.3 Information necessary for scheduling

Participant priority The system must know about the attendance priority of every participant, that is, how important the presence of a participant is for the success of the meeting. The priority enables the meeting initiator to add the notion of "bias" to the scheduling process, which, by itself and if implemented correctly, should be impartial by default.

Participant calendar In order to decide whether or not a participant is free for a given time, the system must have the means to query a participants availability for any given time inside the time span in which the meeting is being scheduled.

Participant preferences Each person has her own unique set of personal and business priorities, preferences and constraints, which are usually much more complex than "free/not free" (Chun et al., 2003), and much work has been dedicated to devise approaches capable of learning a user's preferences (Oh and Smith, 2004; Crawford and Blum, 2009; Crawford and Veloso, 2005a).

It is therefore essential that the meeting scheduling system is aware of such user preferences by being able to accommodate different, isolated preference retrieval methods that can be easily replaced without modifying other parts of the system. This will allow the the system to reach meeting time consensuses that all participants are satisfied with, depending on the quality of the data retrieved from the preferences module.

3.3.4 Scheduling process

A generic, non-optimized meeting scheduling process extracted from the scenario is depicted in Figure 3.1.

In this process, the meeting initiator, after having gathered all information relevant to the meeting, picks any meeting proposal. This proposal is then sent to an invitee (the order in which invitees are selected is not relevant to the process). Said invitee now evaluates the meeting proposal and decides to send one of the following responses:

- A **rejection** is sent in case the invitee does not want to attend the meeting. In that case, that invitee is deleted from the list of invitees by the meeting initiator and does not receive any more messages for the event being scheduled.
- A **counter-proposal** is sent in case the proposal can not be accepted, but the invitee wishes to attend the meeting (she now is considered a meeting participant). Given that the proposal that has been sent is now no longer feasible, a new meeting proposal is picked by the meeting initiator (which improves his proposal by the knowledge gained in the last scheduling “round”), and the scheduling process is started over.
- An **approval** is sent, notifying the meeting initiator that the proposed meeting time is viable for the participant.

If a counter-proposal is received, the process is started over, beginning with the selection of a new proposal, based on the information that has been gathered in previous iterations. After all participants have accepted a proposal, the meeting time that was decided is communicated to all participants. In case the meeting needs to be re-scheduled, the entire meeting scheduling process starts over.

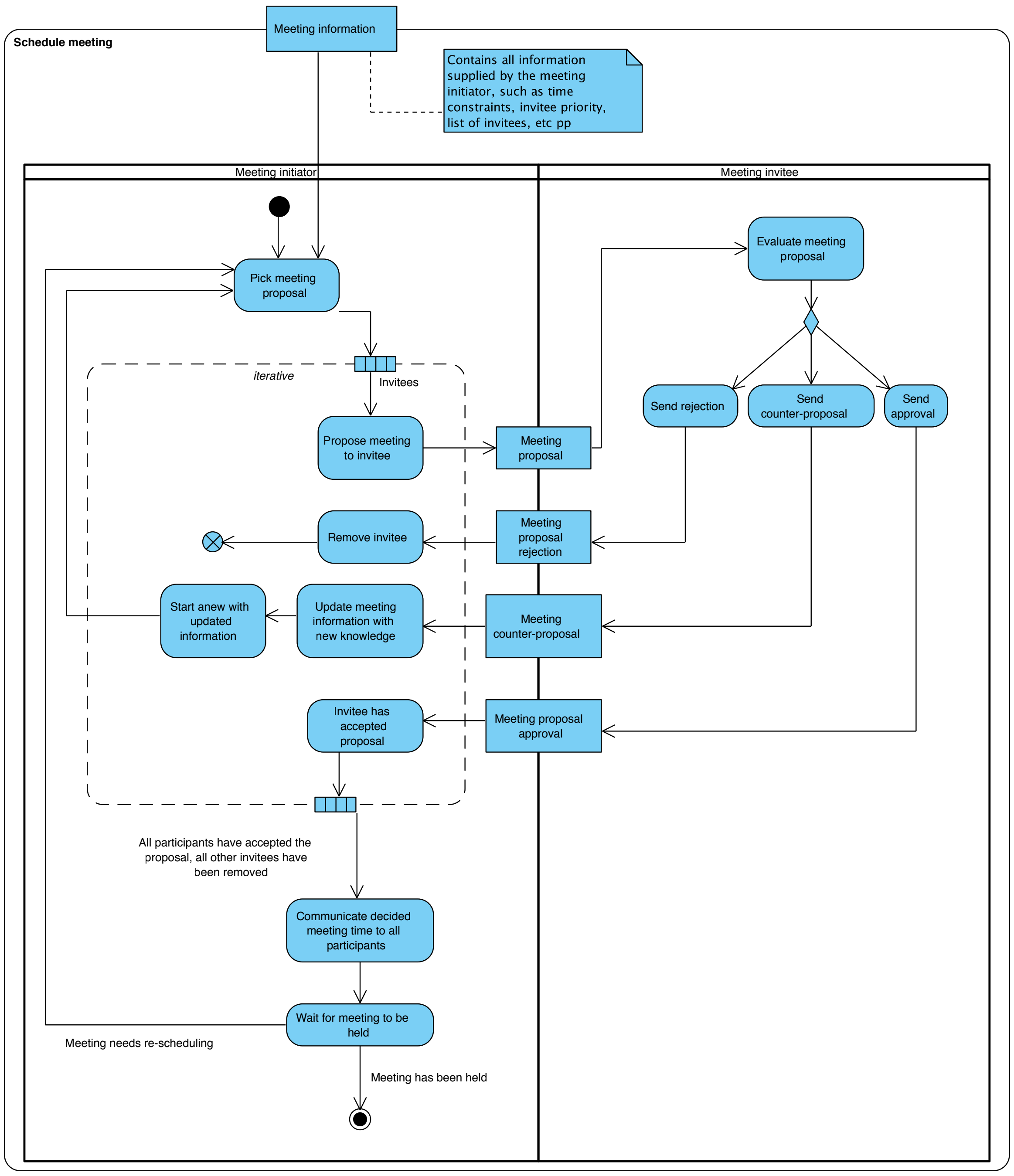
Schedule meeting
Meeting information
Contains all information supplied by the meeting initiator, such as time constraints, invitee priority, list of invitees, etc pp
Meeting initiator
Meeting invitee
Pick meeting proposal
iterative
Invitees
Propose meeting to invitee
Meeting proposal
Evaluate meeting proposal
Send rejection
Send counter-proposal
Send approval
Remove invitee
Meeting proposal rejection
Start anew with updated information
Update meeting information with new knowledge
Meeting counter-proposal
Invitee has accepted proposal
Meeting proposal approval
All participants have accepted the proposal, all other invitees have been removed
Communicate decided meeting time to all participants
Wait for meeting to be held
Meeting needs re-scheduling
Meeting has been held

Figure 3.1: Scheduling process represented in UML activity diagram notation

3.3.5 Scheduling conflicts

When scheduling a meeting, with increasing number of participants, the probability that the intersection of free meeting slots of all participants will yield any result decreases exponentially. As an example, the average time spent in meetings per week in a high-technology coorporation by management level staff is 8.4 hours, the average meeting duration being 60 minutes (Romano and Nunamaker, 2001). Assuming a 40 hour week (i.e. 5 days of 8 hours), this accounts for 21% of their total work time.

As an example, a meeting of one hour should be scheduled in the described environment. Assuming that said meeting is the “last” meeting of the week to be scheduled, that is, statistically 7.4 hours (the total average of 8.4 hours minus the one hour meeting) have already been scheduled, and further assuming that the meetings that have already been scheduled are distributed evenly accross the week, the probability of a free time slot being available to seven people (the average number of meeting participants in said corporation) is about 24%, and only 5% for a larger meeting of 14 participants, as described by the following function:

$$P_{\text{free}}(\text{participants}) = \left(1 - \frac{7.4h}{40h}\right)^{\text{participants}}$$

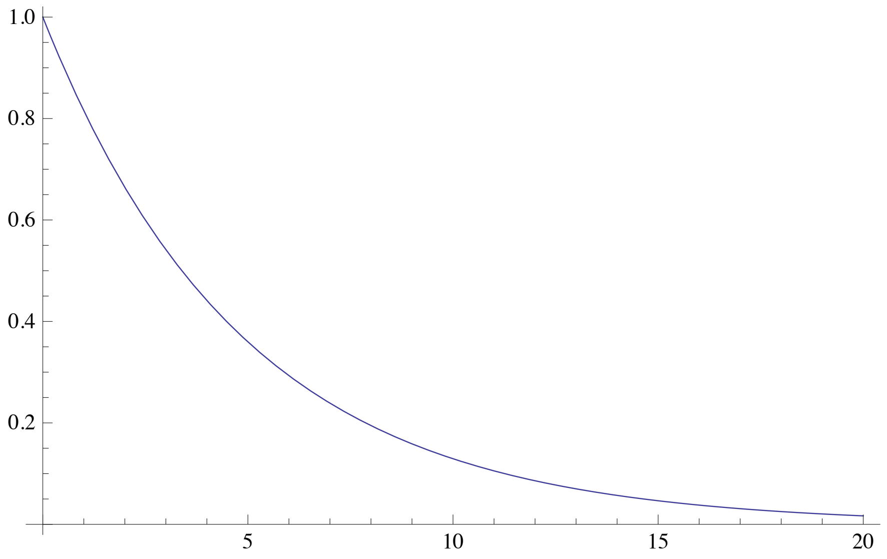

Figure 3.2: Plot of $P_{\text{free}}(\text{participants})$ in $[0, 20]$

However, these probabilites completely disregard every other aspect (except for the meetings) of the participants’ week schedules, such as their personal preferences, lunch breaks, regular work time, etc. Therefore it can be assumed that the actual probability for finding a free meeting slot is much lower. As a result, in order to be of any use in practice, the system must be able to handle scheduling conflicts, for example by re-scheduling existing meetings or by discarding participants of lower priority.

3.3.6 Scheduling fairness

The absence of any bias other than the one explicitly stated by the meeting initiator through the invitee priority must be guaranteed by the system, because otherwise the subset of participants who the system is biased against will be dissatisfied with the scheduling result. However, in this context, fairness does not only refer to the absence of bias, but also to the existence of a guarantee that the preferences of every participant, weighted by its priority, will not be ignored by the scheduling system. In other words, a "fair" scheduling system will always honor the preferences of every participant, even if that means that the overall scheduling result might be less optimal. If an unfair result is explicitly desired, it can be achieved by adjusting the priority of any given invitee to give him an advantage/disadvantage.

3.4 Existing techniques

Meeting scheduling is a member of the class of resource scheduling problems that is known to be computationally intractable, and hence requires heuristics to reduce the computational effort (Sugumaran, 2011). The existing techniques can be divided in two basic categories: **centralized** and **distributed**.

3.4.1 Centralized techniques

Centralized meeting scheduling techniques assume that all information necessary for the meeting scheduling process is available in a central location, at which the meeting scheduling will be executed.

An example of a centralized meeting scheduling system is the one developed by Sugihara et al. (1989), which treats meeting scheduling as a timetable rearrangement problem. Given a timetable of meetings as well as an additional meeting to be scheduled, their system focuses on rearranging the timetable so that the number of persons who have to change their schedule is minimized. This approach, however, completely disregards priorities for meetings and participants as well as participant preferences. Sugumaran et al. (2003) devised a similar approach which mainly differs in their use of the A*-Algorithm to find the best timetable rearrangement solution.

In conclusion, centralized meeting scheduling systems require all data on meetings and participants to be known to the system, which can be difficult to accomplish because the data has to be kept in sync with the users' calendars. It may also be undesirable from a privacy point of view, because people might not be willing to disclose their full calendars to a third party. On the other hand, the meeting scheduling process in a centralized system requires no constant communication between autonomous entities to gather the necessary data, and as such it has the potential to deliver a much faster response.

3.4.2 Distributed approach

The vast majority of work published on meeting scheduling focuses on an approach using autonomous agents to solve the scheduling problem. In this approach, the users' calendar and preferences are located at a location of their choice, together with one or more autonomous agents which are in charge of communicating with other such agents in order to schedule a meeting.

A non-automatic system, i.e. one that aids the meeting scheduling process carried out by the users rather than conducting it by itself, was developed by Biswas and Bhonsle (1992) In their system, which is distributed in the sense that the negotiation software instances are located on many different nodes, participants asynchronously reply to meeting scheduling requests until a meeting consensus has been found. Their work also addresses the question on how the meeting scheduling process can be carried out despite inaccuraccies in users' calendars and the temporary inavailability of some of the system's nodes.

Mattern and Sturm (1989) developed a truly distributed and automatic meeting scheduling system based on the notion of agents, taking into account important aspects of meeting scheduling such as conflict resolution and participant preferences. The resulting prototype system also featured a GUI resembling traditional paper calendars in order to simplify use of the system. While their work often mentions meeting time proposal and conflict resolution heuristics, these are never explained. Bui et al. (1995) take the inverse approach, ommitting most of the details on their meeting scheduling system itself while focusing on the formalization of the scheduling problem as well as their scheduling heuristics.

In summary, distributed meeting scheduling has the advantage that the location of a users' information can be chosen by him and must not be synchronized with a centralized system, protecting the users' privacy. The distributed nature also makes it possible to implement a system that works despite the failure of single nodes, given its absence of a single point of failure. As a result, however, distributed scheduling systems require a large amount of communication between the distributed entities in order to reach a meeting scheduling consensus. The reason why such a large amount of the work on meeting scheduling uses multi-agent system is the fact that it is very easy to represent the problem as it is carried out by humans, by replacing them with autonomous agents, which results in an abstraction that is easily comprehensible.

3.5 Conlusion

As the scenario analysis shows, meeting scheduling is a highly distributed process in its nature, in the sense that much communication between different autonomous entities is necessary in order for it to be successful when carried out manually. It is important to note that, in the context of meeting scheduling, “successful” does not mean “finding a *viable* time”, but “finding a *satisfactory* time for all participants”. The problem with finding such a time is that it is difficult to estimate how satisfied a user might be with a given proposal, making it necessary to have very detailed information on the user’s preferences in order to really reach a satisfactory result. Previous work on the subject has never focused on a flexible architecture for accomodating different scheduling techniques, always resulting in very specialized systems of which many have been developed in the past. Therefore, the aim of this thesis is to provide a base multi-agent architecture and negotiation protocol to be used when implementing future meeting scheduling systems, resulting in extensible implementations that can be interfaced with one another. Said system must be able to accomodate implementations which meet all of the following requirements:

The system must:

1. be architecturally based on the multi-agent paradigm
2. provide the meeting initiator with the means to trigger a new meeting scheduling process, supplying the necessary data to the system
3. be able to schedule meetings for any number of participants, resolving scheduling conflicts if necessary
4. provide the means to easily exchange the implementational units that define the system’s scheduling behavior.
5. make it possible to exchange the methods of calendar data and user preferences retrieval

The scheduling protocol must:

1. cover the entire meeting scheduling process
2. not make any assumptions about the heuristics that are used to carry out the scheduling process

4 Design

In this chapter, a system capable of multi-agent meeting scheduling based on the requirements gathered in chapter 3 is devised, an overview of which can be observed in Figure 4.1. It consists of three major components: the Agent Communication Layer, the Personal Agent and the Meeting Agent.

The Agent Communication Layer, which provides the living environment as well as the basic language for the autonomous agents that will be in charge of scheduling the meetings, is briefly explained in the first section, followed by an explication of the personal- and meeting agents' design. Lastly, the agent dialogues (i.e. the meeting scheduling protocol) is described, laying the foundation for the prototype implementation to be developed in the next chapter.

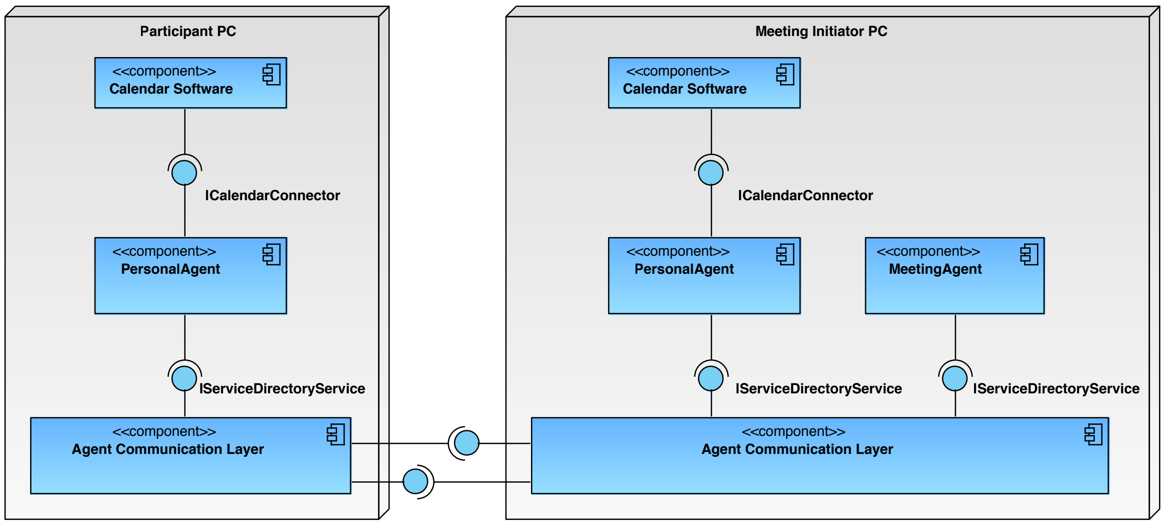

Figure 4.1: UML deployment diagram of the scheduling system

4.1 Agent Communication Layer

The agent communication layer is the foundation for agent discovery and communication throughout the meeting scheduling system, and is designed as a subset of the FIPA Abstract Architecture, which aims to provide a standard for inter-operable agent applications and systems (Foundation for Intelligent Physical Agents, 2002a, p.3). It consists of four components as shown in Figure 4.2, and is capable of providing the following answers for every agent it contains:

- Who am I? - Agent identity
- Where am I? - Agent location
- What am I capable of? - Agent capabilities
- How do I communicate with others? - Agent communication

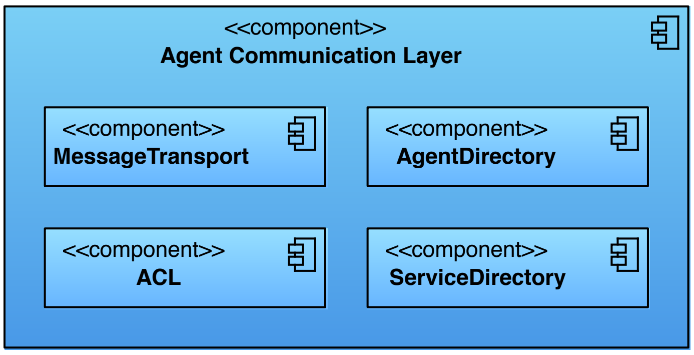

Figure 4.2: Component view of the Agent Communication Layer

These components, which cover all of the questions mentioned above, will be explained briefly in this section. For an in-depth explanation of the components the following FIPA specifications should be consulted:

- Abstract Architecture Specification (Foundation for Intelligent Physical Agents, 2002a)
- ACL Message Structure Specification (Foundation for Intelligent Physical Agents, 2002b)
- Communicative Act Library Specification (Foundation for Intelligent Physical Agents, 2002c)

4.1.1 Agent Directory Service

The agent directory service enables an agent to search for other agents it wishes to interact with. Upon agent registration, an agent directory entry is created, which must at least contain a globally unique name and an agent locator for the agent. It may also contain additional attributes that provide more information about an agent, such as information about the capabilities or the type of an agent.

The agent locator consists of a set of transport descriptions, which in turn are made up of a transport type and a transport address. Depending on what transports a given agent supports, it can then choose through which transport to communicate with an agent retrieved from the agent directory service. An example of an agent directory entry can be observed in Listing 1.

Listing 1: Example of an agent directory entry in JSON representation

```
{
    "agent-name": "public@x3ro.de",
    "agent-locator": [
        {
            "transport-type": "akka.actor.Actor",
            "transport-specific-address": "akka://my-system/user/public@x3ro.de"
        }
    ],

    // The entry may contain additional descriptive attributes.
    "agent-attributes": {
        "agent-type": "personal-agent",
        "rescheduling-support": "true"
    }
}
```

4.1.2 Service Directory Service

The service directory service provides agents with the ability to discover other services, such as the Message Transport Service and the Agent Directory Service. An agent must bootstrap agent communication by means of the Service Directory Service, which is therefore a mandatory parameter when creating a new agent. Analogous to the agent directory, the service directory contains service directory entries. An entry referencing an agent directory service is illustrated in Listing 2.

Listing 2: Example of a service directory entry in JSON representation

```json
{
    "service-name": "dir-service-1",
    "service-type": "org.fipa.standard.service.agent-directory-service",
    "service-locator": [
        {
            "signature-type": "akka.actor.TypedActor",
            "service-signature": "net.jenss.thesis.agent.AgentDirectory",
            "transport-specific-address": "akka://agent-system/user/dir-service-1"
        }
    ]
}
```

4.1.3 Message transport

The message transport is possibly the most complex part of the agent subsystem, as it realizes the actual communication between the agents, and as such is in charge of intricate tasks such as guaranteeing successful message transportation as well as message validity. Given the inherent complexity of this module as well as the wide range of implementations that are possible, the message transport will not be further discussed, as such a digression would go beyond the scope of this thesis. An abstract explanation of the message transport may consulted in Section "4.5 - Agent Messages" of Foundation for Intelligent Physical Agents (2002a).

4.1.4 Agent Communication Language (ACL)

The agent communication language provides the basic structure and vocabulary for the language that the agents communicate with. FIPA defines a conversation between two agents as a series of “communicative acts”, each identified by its “performative” or type of the communicative act. The performative is the only mandatory parameter that every communicative act must contain. The following optional parameters are also important for the purpose of this thesis:

- Content
- Sender and Receiver, in order to route the messages to the correct destination and to know which agent gave the received answer.
- The Conversation-Id is used to differentiate responses belonging to multiple meeting scheduling processes running in parallel.

The FIPA Communicative Act Library Specification (Foundation for Intelligent Physical Agents, 2002c) provides a number of generic, well defined performatives, a subset of which is used in the meeting scheduling protocol. This subset is subsequently presented along with a brief description on the semantics of the performatives.

Performative	**Semantics**
Accept	The action of accepting a previously submitted proposal
Cancel	The action of an agent informing another agent that the former does not wish the latter to continue performing a given action
Confirm	The sender agent informs the receiving agent that a previously sent proposition or request is true
Disconfirm	The sender agent informs the receiving agent that a previously sent proposition or request is false
Failure	The action of an agent informing another agent that an action or request has failed
Inform	The sender informs the receiver that a given proposition is true
Propose	The act of submitting a proposal to another agent, proposing to perform some action
Request	The sender requests the receiver to perform an action

Table 4.1: FIPA ACL performatives used in this works’ meeting scheduling protocol

These performatives form the foundation vocabulary for the meeting scheduling protocol presented later in this chapter.

4.2 Agents

The agents within the meeting scheduling system are autonomous entities, which, in communication with each other, are able to negotiate a meeting time for any given meeting, as long as a possible time for such a meeting exists under consideration of every agents' owner preferences. The agents are divided into two types with different reponsabilities: **Meeting Agents** and **Personal Agents**, the former being in charge of conducting the meeting scheduling process (thus being the active component) whereas the latter must simply respond to questions that the meeting agent asks. This division is based mainly on the work of Chun et al. (2003) as well as other systems which feature a similar component division, such as Mattern and Sturm (1989).

The reasons for this design decision are manifold. For one, the division allowed a very cohesive design of both agents, resulting in a clean separation of meeting scheduling (active) and informative (passive) functionality. The eventuating components are smaller, thus easier to understand, modify and maintain.

An active component conducting the negotiation process is also necessary in order to reduce the negotiation complexity. As an example of a negotiation process not featuring such a component, one might imagine trying to schedule a meeting of several participants by sending emails, without a single person managing the proposals. The participants could first reach an agreement in groups of two, and then propose their agreed time to a third participant, and so on. Such an approach, however, will lead to an incredibly large amount of emails being generated (and time taken), given the fact that every scheduling conflict, of which there will be many (Section 3.3.5 Scheduling conflicts), will result in the meeting scheduling process beginning anew.

The central role of the meeting agent as the active component in charge of managing the scheduling process, inquiring information from all participants of the event to be scheduled, can be seen as an analogy to the very similar meeting scheduling approach by humans. This similarity allows the construction of a meeting scheduling protocol that is also similar to the way humans interact when trying to find a time for a meeting.

4.2.1 Personal agent

This section covers the personal agent's architecture. The personal agent acts as a passive representative of a single user and could, for example, run as a service on that user's PC (thus the name "personal"). As such, it must have knowledge of the user's preferences and his current calendar, enabling it to initiate a meeting scheduling process (triggered explicitly by the user) and to automatically respond to meeting proposals and other requests sent by the meeting agent, acting as an exclusively passive component. By only responding to specific meeting proposals the user's calendar data and preferences are kept as private as possible. The personal agent is a passive component because it only responds to inquiries made by the meeting agent, but never initiates a communication by itself. An architectural overview of the personal agent can be seen in Figure 4.3.

The personal agent has two interfaces that connect it with its environment, the `IPersonalAgentConfiguration` and the underlying agent system bootstrapped by the `IServiceDirectoryService`.

Component configuration

The `IPersonalAgentConfiguration` interface allows configuration of all sub-components inside the personal agent, hence providing the flexibility to configure custom sub-component implementations without actually exposing them to the environment of the personal agent. Since it should be possible to configure all sub-components, those must implement a common configuration interface: `IConfiguration` (Listing 3).

The `IPersonalAgentConfiguration` interface, in contrast to `IConfiguration`, allows to address the component of which a configuration parameter should be set or retrieved (as in Listing 4, which shows the target component mechanism using an enumeration).

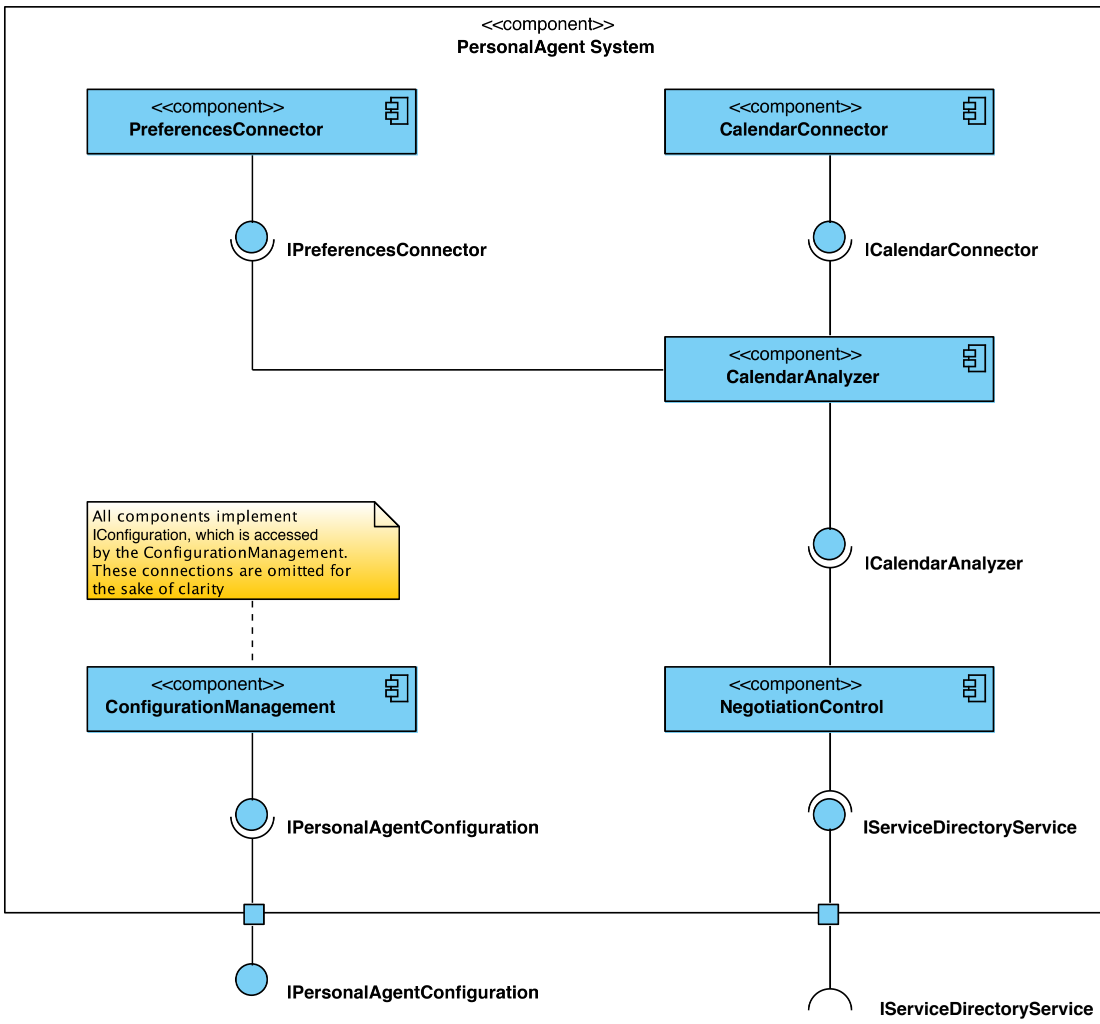

Figure 4.3: Overview of the personal agent's architecture

Listing 3: The IConfiguration Interface of the meeting scheduling system

```
trait IConfiguration {
    /**
     * Sets a configuration parameter identified by "name" to the given value.
     * @return FALSE on success, otherwise FALSE
     */
    def set(name:String, value:String):Boolean

    /**
     * Retrieve a configuration parameter identified by "name".
     * @return The value for the given parameter or
     *      None if no such parameter was set.
     */
    def get(name:String):Option[String]
}
```

Listing 4: The IPersonalAgentConfiguration Interface of the Personal Agent System

```
object Component extends Enumeration {
    type Component = Value
    val NegotiationControl, CalendarAnalyzer, [...] = Value
}

trait IConfiguration {
    def set(target:Component, name:String, value:String):Boolean
    def get(target:Component, name:String):Option[String]
}
```

Calendar Analyzer

The calendar analyzer component is responsible for supplying the `NegotiationControl` with information about the user's calendar and preferences. Therefore, all decisions made by the `NegotiationControl` component should be solely based on its configuration and the information received through the `CalendarAnalyzer`. Listing 5 shows the `ICalendarAnalyzer` interface, featuring methods to check if there is any free slot for a meeting, get the best free slot available or rate a given time slot according to the user's preferences.

Listing 5: The ICalendarAnalyzer Interface of the Personal Agent System

```
trait ICalendarAnalyzer {
    /**
     * Retrieves the best free timeslot for this meeting.
     */
    def getBestFreeSlot(
        m:Meeting,
        priorityThreshold:Double
    ):Option[MeetingTimeProposal]

    /**
     * Checks whether or not there is _any_ time slot that fits the needs of the
     * given meeting
     */
    def hasFreeTimeSlot(m:Meeting, priorityThreshold:Double):Boolean

    /**
     * Returns a rating for the given time slot according to the user's calendar
     * and preferences
     */
    def rateTimeSlot(
        m:Meeting,
        mtp:MeetingTimeProposal,
        priorityThreshold:Double
    ):Double
}
```

The `priorityThreshold` parameter defines which meeting slots should be considered free even if there are meetings already scheduled, based on the priority of the meeting (e.g. if threshold were `0.5`, all meetings with a priority of $[0.0, 0.5]$ would be considered free). This allows the personal agent to gradually "relax" its priority preference if no meeting time consensus can be found, at the cost of having to re-schedule an already existing meeting. For example, if a meeting with priority `0.4` is to be scheduled, the personal agent could start by sending proposals for time slots with no meetings scheduled (`priorityThreshold = 0.0`), and if none are found, gradually increase the `priorityThreshold` parameter until a meeting to be re-scheduled is found.

The components supplying information to the `CalendarAnalyzer` are connected via the `IPreferences`- and `ICalendarConnector` interfaces. These are the most essential parts of the personal agent, given that they are the foundation for its reasoning process about time slots, which ultimately decides when a meeting will be scheduled. Depending on the implementation of these components, only a few of the questions that may be answered implicitly by these interface through the satisfaction values are:

- Do I already have a meeting scheduled here, and how important is it to me?
- Would I rather have a break at this time, instead of scheduling a meeting?
- What is my general preference for this time interval?

The `ICalendarConnector` interface (Listing 6) provides information about a user's calendar to the personal agent. It features a method to check whether or not the user has meetings scheduled in a given time interval, and if so, retrieve their priorities (`priorityAt`), and one to retrieve all possible time slots for of a given duration in a given interval (`slotsInInterval`).

Listing 6: The ICalendarConnector Interface of the Personal Agent System

```
trait CalendarConnector {
    /**
     * Returns the priorities of any meeting scheduled in the time range beginning
     * at the "start" instant and lasting as long as "duration". If multiple
     * meetings are scheduled in the given interval, their priorities will be
     * returned in chronological order.
     */
    def priorityAt(start:ReadableInstant, duration:ReadablePeriod):Option[List[Double]]

   /**
    * Retrieves all time slots in the given interval, and the priority of the event
    * scheduled at the time slots, if any. The granularity parameter specifies the
    * steps in which slots are retrieved, for example with a granularity of 15
    * minutes, a slot beginning at every full 15 minutes would be retrieved.
    */
    def slotsInInterval(
        interval:ReadableInterval,
        slotLength:ReadablePeriod,
        granularity:ReadablePeriod
    ):SortedMap[ReadableInstant, Option[Double]]
}
```

The `IPreferencesConnector` interface enables the personal agent to retrieve additional satisfaction information for any given time frame, independently of the calendar information. These preferences might be manually entered by the user (as in Mattern and Sturm (1989)) or, for example, be retrieved from a neural network fed with historical data on scheduled meetings as proposed by Lai et al. (2009). The decoupling of the source of the preference information makes it easy to replace it, or even add multiple such sources which could then be compared. The exact way in which preferences are determined is, however, not a part of this thesis.

Listing 7: The IPreferencesConnector Interface of the Personal Agent System

```
trait IPreferencesConnector {
    /**
     * Retrieves the expected satisfaction for a meeting of length "duration"
     * scheduled beginning at the instant "start".
     */
    def getSatisfaction(start:ReadableInstant, duration:ReadablePeriod):Double
}
```

4.2.2 Meeting agent

The meeting agent is the active component of the meeting scheduling architecture. It is in charge of finding a satisfactory meeting time for a number of participants by inquiring and evaluating information from the personal agents. For every meeting to be scheduled, a new meeting agent is created by the personal agent of the meeting initiator, passing it the necessary information to initiate the scheduling process. A meeting agent only ever schedules a single meeting, but it is not destroyed until the meeting has been actually held. This allows the meeting agent to monitor events that might affect the meeting as well as re-schedule the meeting if need be (Chun et al., 2003), which would be more complex if a new meeting agent would have to be created, given that the existing one still holds all the information on the previously completed scheduling process. An overview of the meeting agent's architecture is shown in Figure 4.4.

The meeting agent features the same configuration method as the personal agent (4.2.1 Component configuration), except for the interface that connects the configuration management component to the environment, which is `IMeetingAgentConfiguration`. The connection to the multi-agent subsystem is also the same as in the personal agent.

In order to explain the components and interfaces that make up the meeting agent, some of the messages of the scheduling protocol need to be covered first, given that they are used in the interfaces as well. These messages are `MeetingTimeProposal`, `MeetingTimeProposals` as well as `Meeting`. The latter is an object that contains all information about a certain meeting that is being scheduled (the exact structure is explained in 4.3 Meeting scheduling protocol). A `MeetingTimeProposal` is

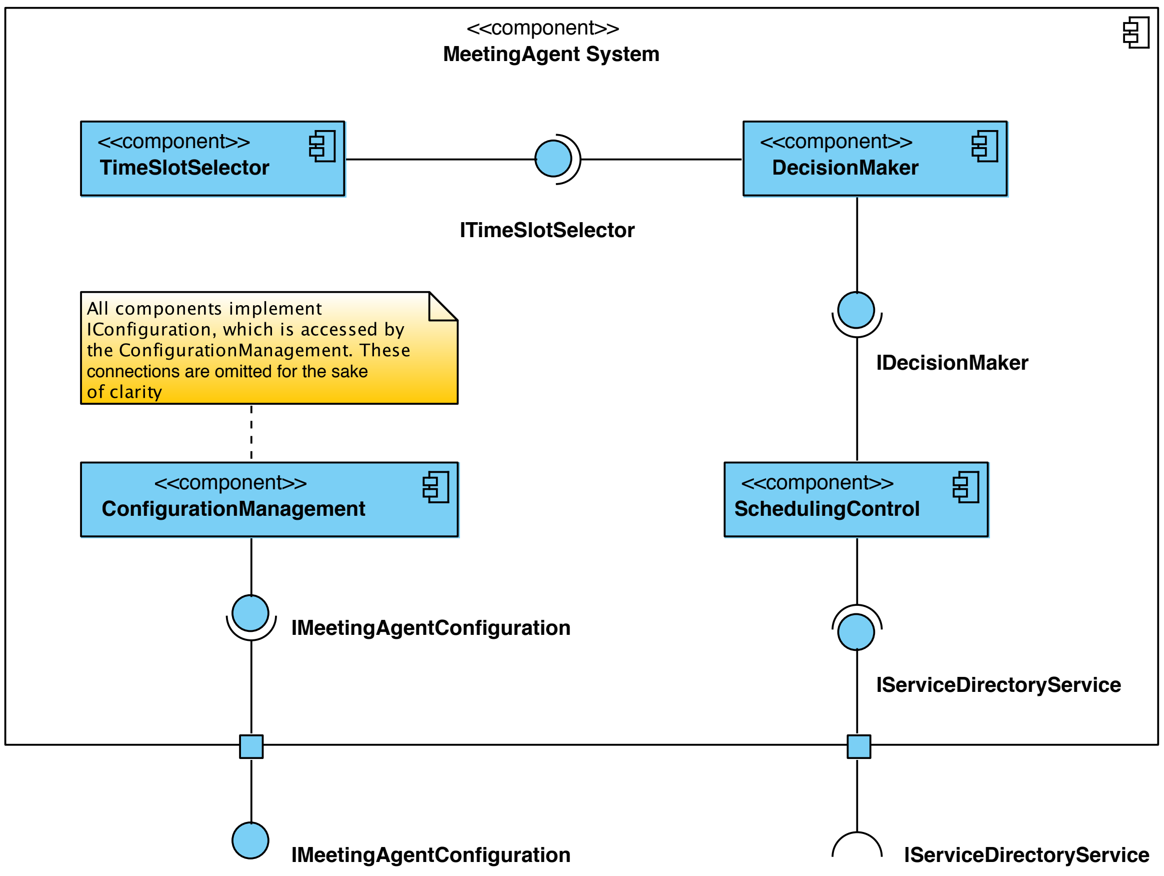

Figure 4.4: UML component diagram of the meeting agent system

made up of start and end date/time as well as a satisfaction value for the meeting time proposal. Lastly, a `MeetingTimeProposals` message is simply a list of `MeetingTimeProposal` objects.

There are also two structures used in the interfaces that are never sent as messages: The `MeetingInformation` object is comprised of all information that has been received by the meeting agent since the beginning of the scheduling process (as can be seen in Listing 8), and it is the basis for all decisions it makes.

The behavior of the meeting agent's communication component, the `SchedulingControl`, is controlled by the `DecisionMaker` component. After every scheduling iteration (i.e. after every participant's personal agent has responded to an inquiry), the `SchedulingControl` component asks the `DecisionMaker` to make a decision on how the scheduling process should continue. Depending on the returned `Decision` object, the `SchedulingControl` then takes the desired action. The `MeetingAgentState` field of the `Decision` object defines the next desired action and takes a value that identifies one of the `SchedulingControl` states shown in the

Listing 8: Structure of the MeetingInformation object

```scala
type ProposalMap = Map[AgentDirectoryEntry, MeetingTimeProposals]
type ProposalList = List[MeetingTimeProposal]
type MessageList = List[CommunicativeAct]

class MeetingInformation(
  meeting:Meeting,
  messageHistory:MessageMap,
  messagesSinceLastDecision:MessageMap,

  // The number of responses expected for the current scheduling iteration. This
  // field is necessary to see if a given iteration has been completed, as otherwise
  // the meeting agent would always have to wait until the iteration timeout is reached
  // until the next decision can be made.
  expectedResponses:Int,
  proposals:ProposalMap,
  acceptedProposals:ProposalMap,
  confirmedReservations:Map[ActorRef, List[Reservation]]
)
```

diagram in Figure 4.5. Along with the next step that the `SchedulingControl` should take, the `Decision` object also contains the messages to be sent specifically to one recipient (the `addressed` field) and to be broadcasted to all participants (the `broadcast` field). The `SchedulingControl` will then send these messages, take the designated state and, after completion, ask the `DecisionMaker` to make the next decision. The `DecisionMaker` itself bases part of its decision on the data analysis supplied by the `TimeSlotSelector` component, which tells the `DecisionMaker` which of a given set of time slots is the most favorable. Its interface is comprised of a single method `selectBestTimeSlot` which receives a set of proposals and returns the best one (as can be observed in Listing 9).

Listing 9: The ITimeSlotSelector interface which controls selection of the best time slots

```scala
trait ITimeSlotSelector {
    /**
     * Selects the best meeting time proposal (based on satisfaction ratings)
     * of a set of proposals.
     */
    def selectBestTimeSlot(proposals:ProposalMap):MeetingTimeProposal
}
```

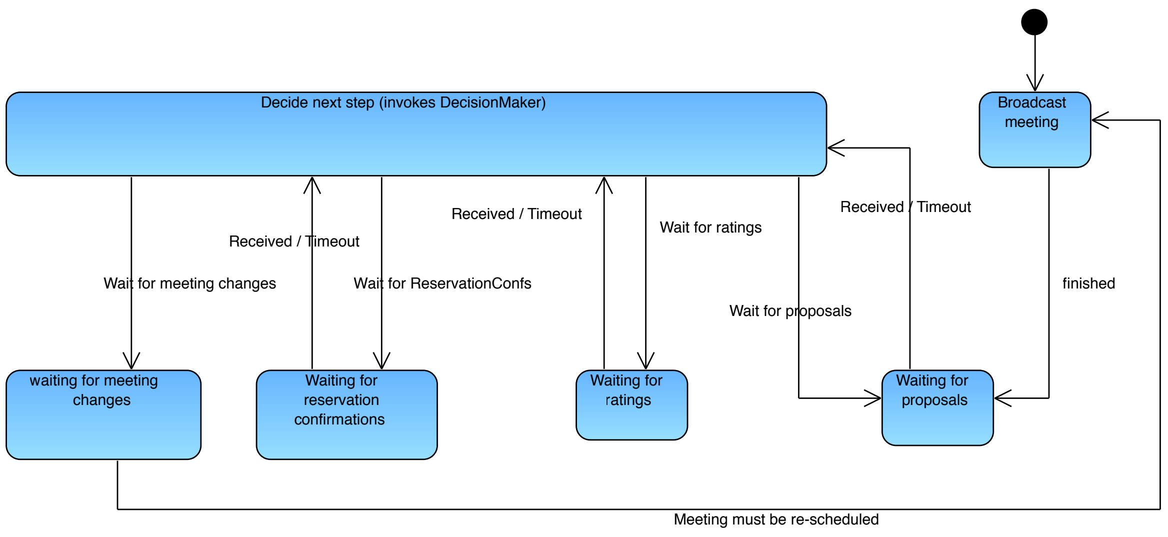

Figure 4.5: Diagram of the states that the SchedulingControl component can take after a decision has been made by the DecisionMaker

Listing 10: The IDecisionMaker interface that controls the SchedulingControl's behavior

```
class Decision(
  nextState:MeetingAgentState,
  broadcast:Option[CommunicativeAct],
  addressed:Option[Map[AgentDirectoryEntry, CommunicativeAct]]
)

trait IDecisionMaker {
  /**
   * Evaluates the current state of meeting time proposals in order to decide what action
   * should be taken next.
   */
  def evaluate(mi:MeetingInformation):Decision
}
```

4.3 Meeting scheduling protocol

In this section, a generalized distributed meeting scheduling process is depicted, based on the FIPA Communicative Acts described in 4.1.4 Agent Communication Language (ACL). The messages being sent by the agents are defined and subsequently put into context of the scheduling process carried out by the “Initiator Agent” (IA), the “Meeting Agent” (MA) and the “Participant Agent” (PA) as illustrated in Figure 4.6.

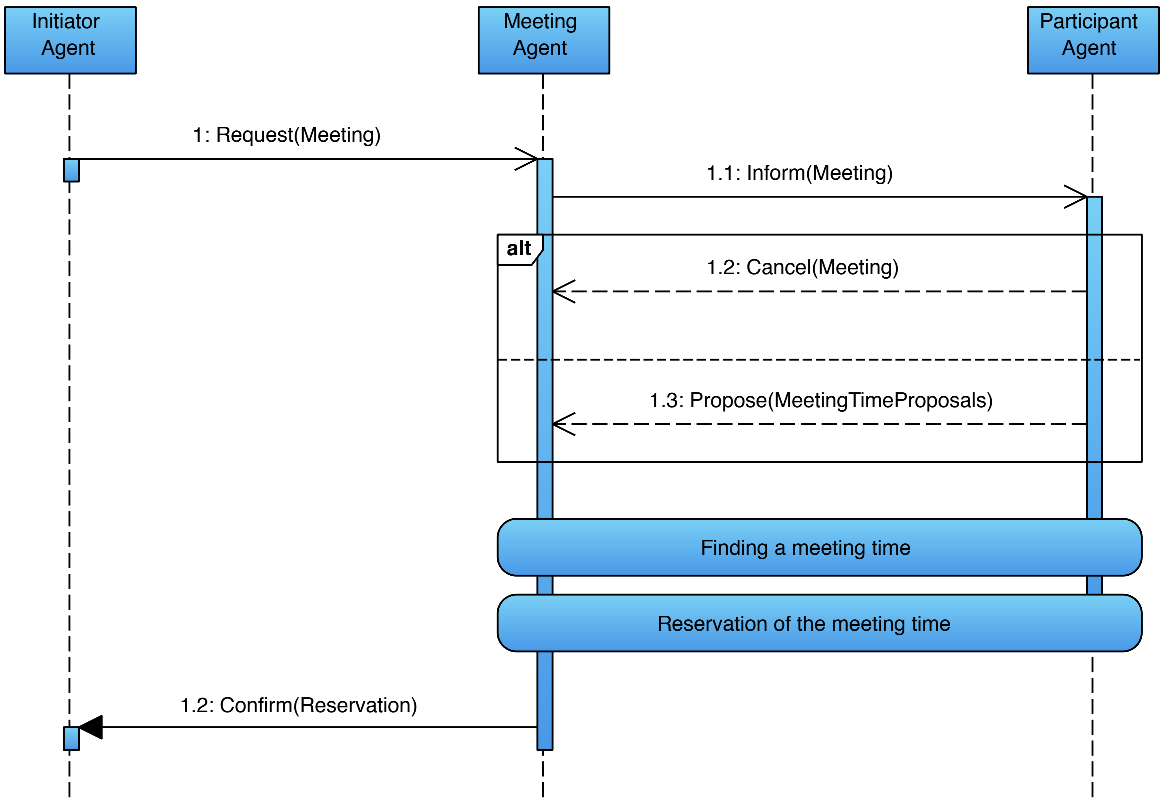

Figure 4.6: Overview of the meeting scheduling process

4.3.1 Initiation phase

The meeting scheduling process is initiated with a `Request(Meeting)` message being sent by the IA to the MA and is structured as shown in Listing 11.

The UUID[1] is expected to be unique within the universe of scheduled meetings

[1] **Universally Unique Identifier**: An identifier that has a unique value within some defined universe.

Listing 11: A Meeting object must contain all information necessary to schedule a meeting

```
Meeting(
    uuid:String,
    duration:Duration,
    whenToSchedule:Interval,
    priority:Double,
    participants:List[AgentDirectoryEntry],
    metadata:Map[String, Any],
    initialProposals:List[MeetingTimeProposal],
    timezone:DateTimeZone
)
```

by the system, so that any meeting scheduling request may be uniquely identified. A `Meeting` request must contain information on the duration of the meeting to be scheduled (e.g. 2 hours), the time interval in which the meeting shall take place (e.g. between 2012-12-01 and 2013-01-15) and the priority given to the meeting by the owner of the IA, ranging from `0.0` (lowest) to `1.0` (highest). It must also hold a map of participants invited, mapping each to their respective attendance priorities.

Additionally, the IA may submit a list of initial `MeetingProposals` that should be proposed to the invitees before any proposal generated by the MA. Finally, it may contain meta-information on the meeting stored in the `metadata` field. However, there is no guarantee that the MA or the PA will recognize or abide by any meta-information submitted. An example of metadata supplied is `Map("number-of-proposals" → 4)`, whose semantics *could* be that every PA should respond with four meeting proposals when prompted.

The MA then informs all PAs of the upcoming meeting scheduling process, which the PAs either "Cancel", discarding them from the process, or accept, by responding with a list of at least one of their most favorable time proposals for the meeting at hand (note that there is no definition on *how many* time proposals must be sent, this is decided by every agent individually).

All PAs either cancelling or accepting a meeting scheduling request concludes the initiation phase.

4.3.2 Finding and reserving a meeting time slot

After successful initiation, the MA (in collaboration with all PAs) tries to find a mutually acceptable time slot to schedule the meeting, as shown in Figure 4.7.

One run of the meeting time search process is exemplified in the following conversation between Alice, representing the Meeting Agent, as well as Bob and Carol, representing two Invitee Agents. For the purpose of this example it is assumed that Alice speaks directly to Bob and Carol, that is, only the addressed person is able to hear what Alice says and only Alice is able to hear their responses.

```
(01) Alice: Bob and Carol, would you like to meet on Monday at 19:00?
(02) Bob  : No, I can’t make it on Monday, but I could make it on Tuesday at 18:00.
(03) Carol: Yes, I could make it on Monday at 19:00.
(04) Alice: Carol, how about Tuesday at 18:00?
(05) Carol: Tuesday at 18:00 would be okay.
(06) Alice: Bob and Carol, would you like to meet on Tuesday at 18:00?
(07) Bob  : Yes, I could make it on Tuesday at 18:00.
(08) Carol: Yes, I could make it on Tuesday at 18:00.
(09) Alice: Bob and Carol, please reserve Tuesday at 18:00 for the meeting.
(10) Bob  : I’ve reserved Tuesday at 18:00.
(11) Carol: I’ve reserved Tuesday at 18:00.
(12) Alice: Bob and Carol, the meeting is scheduled for Tuesday at 18:00.
```

That conversation is equivalent to the following messages being sent from the MA (Alice) to the PAs (Bob and Carol) and back:

#	Sender	Recipient	Message
1	Alice	Bob, Carol	Propose(MeetingTimeProposal)
2	Bob	Alice	Propose(MeetingTimeProposal)
3	Carol	Alice	Accept(MeetingTimeProposal)
4	Alice	Carol	Request(MeetingTimeProposalRating)
5	Carol	Alice	Inform(MeetingTimeProposalRating)
6	Alice	Bob, Carol	Propose(MeetingTimeProposal)
7	Bob	Alice	Accept(MeetingTimeProposal)
8	Carol	Alice	Accept(MeetingTimeProposal)
9	Alice	Bob, Carol	Request(Reservation)
10	Bob	Alice	Confirm(Reservation)
11	Carol	Alice	Confirm(Reservation)
12	Alice	Bob, Carol	Confirm(Reservation)

Table 4.2: Example of meeting scheduling message sequence

The time slot search begins with the MA selecting a `MeetingTimeProposal` (see Listing 12 for a structural view of the mentioned messages) to propose to all PAs. If all PAs accept the proposal, the reservation process is initiated, otherwise the PAs respond with a list of counterproposals. After a response from all PAs has been received by the MA, the newly acquired data is evaluated. While evaluating, the MA might need additional data that it can request from specific PAs by sending `Request(MeetingTimeProposals)` or `Request(MeetingTimeProposalRating)` messages. After the evaluation, the MA selects a new proposal which is then once again proposed to all PAs. This process is repeated until either all PAs have accepted the proposal or no more meeting proposals remain.

After a proposal has been accepted by all PAs, a three-way handshake reservation is carried out in order to assure that a time slot is not accidentally assigned to multiple meetings. The MA sends a `Request(Reservation)` messages to all `PAs`, which reply either with a `Confirm(Reservation)` in case the time slot is still available and not reserverd, or a `Disconfirm(Reservation)` otherwise. If all PAs confirm the reservation, the MA then broadcasts a `Confirm(Reservation)` message itself, giving final confirmation over the meeting time and thus concluding the meeting scheduling process. The reservation process is additionally depicted in Figure 4.8.

Listing 12: Structure of the messages sent in the time slot search process

```
class MeetingTimeProposal(
    begin:ReadableDateTime,
    end:ReadableDateTime,
    satisfaction:Double
)

class MeetingTimeProposals(
    meeting:Meeting,
    proposals:List[MeetingTimeProposal]
)

// The rating message does not contain additional information, and thus only
// exists to distinguish between proposal and rating request.
class MeetingTimeProposalRating(
    proposal:MeetingTimeProposal
)

case class Reservation(
    meeting:Meeting,
    proposal:MeetingTimeProposal,
    reservedUntil:ReadableDateTime
)
```

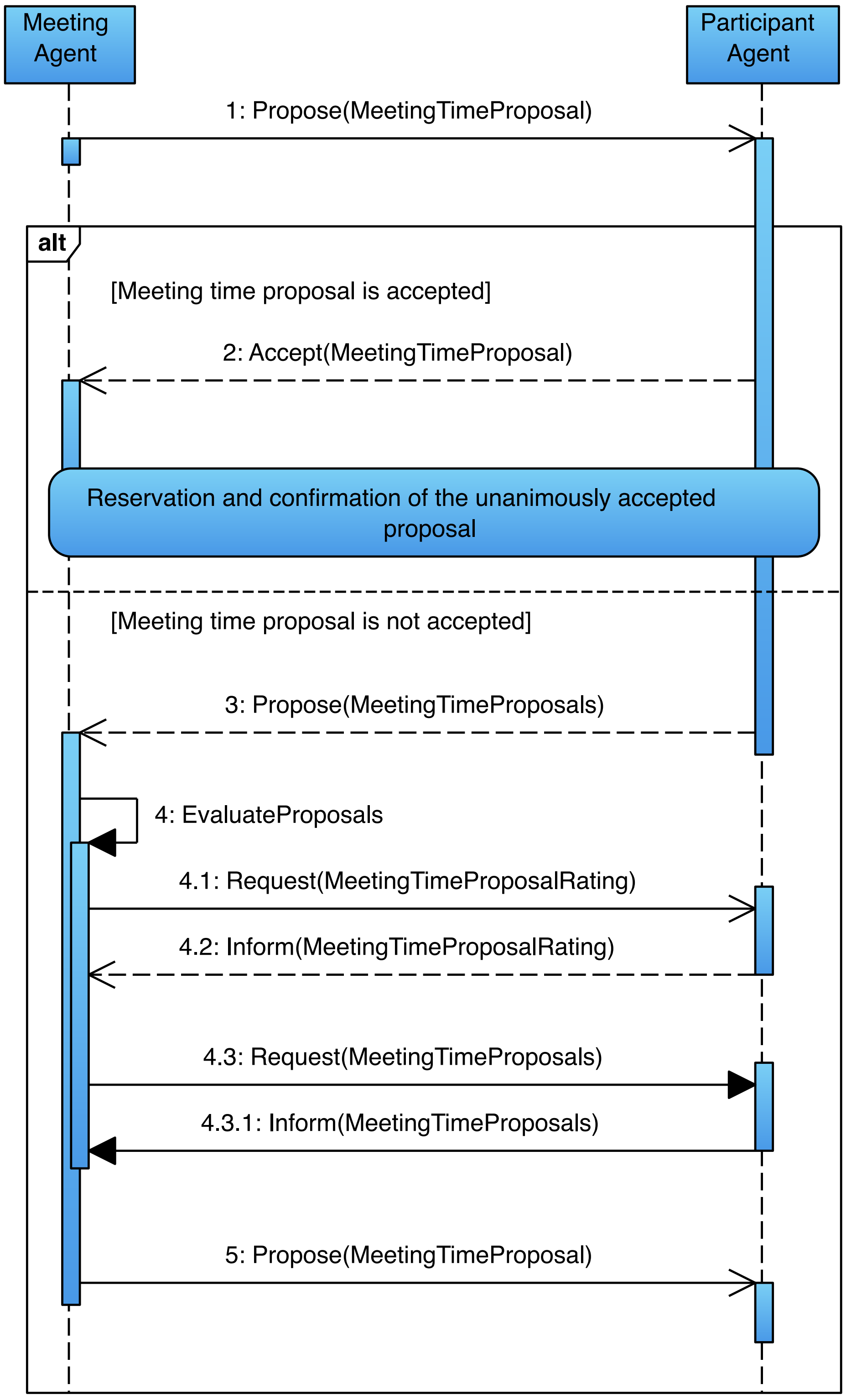
Meeting Agent
Participant Agent
1: Propose(MeetingTimeProposal)
alt
[Meeting time proposal is accepted]
2: Accept(MeetingTimeProposal)
Reservation and confirmation of the unanimously accepted proposal
[Meeting time proposal is not accepted]
3: Propose(MeetingTimeProposals)
4: EvaluateProposals
4.1: Request(MeetingTimeProposalRating)
4.2: Inform(MeetingTimeProposalRating)
4.3: Request(MeetingTimeProposals)
4.3.1: Inform(MeetingTimeProposals)
5: Propose(MeetingTimeProposal)

Figure 4.7: UML sequence diagram of the meeting time search protocol

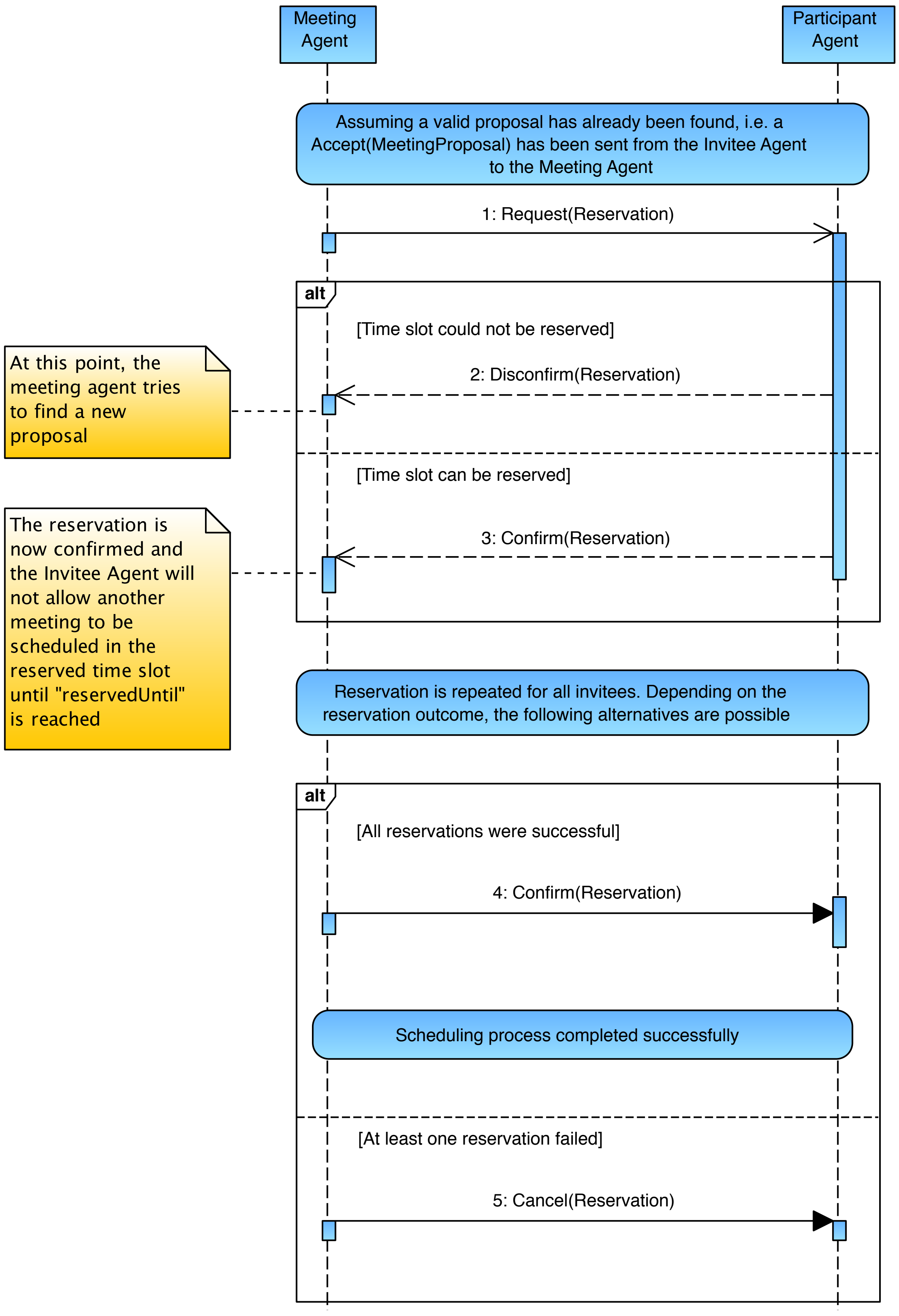
Meeting Agent
Participant Agent
Assuming a valid proposal has already been found, i.e. a Accept(MeetingProposal) has been sent from the Invitee Agent to the Meeting Agent
1: Request(Reservation)
alt
[Time slot could not be reserved]
2: Disconfirm(Reservation)
At this point, the meeting agent tries to find a new proposal
[Time slot can be reserved]
3: Confirm(Reservation)
The reservation is now confirmed and the Invitee Agent will not allow another meeting to be scheduled in the reserved time slot until "reservedUntil" is reached
Reservation is repeated for all invitees. Depending on the reservation outcome, the following alternatives are possible
alt
[All reservations were successful]
4: Confirm(Reservation)
Scheduling process completed successfully
[At least one reservation failed]
5: Cancel(Reservation)

Figure 4.8: UML sequence diagram of the meeting time slot reservation process

4.3.3 Possible concurrency issues

By performing a three-way handshake when reserving the meeting time slot after a meeting proposal has been confirmed, the scheduling protocol makes sure that no two meetings are accidentally scheduled for the same time. When multiple meeting scheduling processes led by different meeting agents are performed in parallel however, this might lead to competitive behavior between multiple agents, which can result in non-optimal solutions or no solution at all being found by the agents. For the purpose of this thesis and the prototype implementation, it will be assumed that only a single meeting scheduling process will run for every time interval. Given that this is not a solution to the problem, subsequent work based on this thesis might concern itself with the extension of the existing technique allowing an arbitrary number meetings to be scheduled for any given time interval, for example by enabling the meeting agents to communicate among one another in order to coordinate parallel meeting scheduling process.

4.4 Time slot selection

In order to provide a meeting scheduling result that satisfies all attendees, an approach for determining overall user (de-)satisfaction will be devised in this section. This technique is specifcally designed as an example of how data might be evaluated in a multi-agent meeting scheduling system, specifically the `TimeSlotSelector` component of the meeting agent, and does not aspire to be a unique contribution, given that similar topics have been extensively covered in both the field of mathematics and computer science (non-linear optimization, dynamic programming, etc.)

As an example, the satisfaction data on four time slots given by three users' personal agents (as shown in Table 4.3) will be examined. The satisfaction data that would serve as input to the subsequently devised technique is a result of the reasoning process of the personal agents as it is covered in 4.2.1 Calendar Analyzer.

	Slot 1	Slot 2	Slot 3	Slot 4
User 1	1.0	0.0	0.1	0.3
User 2	0.7	0.95	0.05	0.5
User 3	0.1	0.0	0.2	0.6

Table 4.3: Satisfaction data to be examined

The satisfaction values in the cells shown in Table 4.3 lie in the interval $[0, 1]$ and have the following semantics:

- 1.0 $\rightarrow$ very satisfied with the proposal
- 0.5 $\rightarrow$ indifferent
- 0.0 $\rightarrow$ very unsatisfied with the proposal

To demonstrate how the different techniques affect the selection outcome, each of the timeslots has a distinct characteristic:

- In slot 1 two users are satisfied, while one is very unsatisfied.
- In slot 2 a single user is very satisfied while two are completely unsatisfied.
- In slot 3 all three users are mostly unsatisfied.
- In slot 4 all three users are close to indifference.

4.4.1 Satisfaction data evaluation

As a first step, an arithmetic mean was applied to the data. If the mean value was decisive for the selection, Slot 1 would be chosen as the highest rated slot, which sacrifices the satisfaction of User 3 almost entirely. This effect can be seen in Table 4.4, where Slot 1 yields the highest mean value.To prevent this from happening, the concept of **benefit** is introduced, which is defined as the distance of a given satisfaction value from 0.5, i.e. user indifference:

$$Benefit(x) = x - 0.5$$
$$\forall x \in [0,1] : Benefit(x) \in [-0.5, 0.5]$$

Given that the benefit yields positive and negative values, an arithmetic mean is not applicable. Because of this, the **negative benefit average** (NBA) and the **positive benefit average** (PBA) is used, where the former is the average of the benefit values below zero and the latter averages all values equal to and above zero, as can be seen in Listing 13.

Listing 13: Positive and negative benefit average calculation

Precondition: Slot must be a bag of n satisfaction values

```
function PositiveBenefitAvg($\mathrm{Slot}$)
    $\delta \leftarrow \varnothing$
    for $i \leftarrow 1$ to $n$ do
        $\lambda \leftarrow \mathrm{Benefit}(\mathrm{Slot}_i)$
        if $\lambda \geq 0$ then
            $\delta \leftarrow \delta \cup \{\lambda\}$
    return $\frac{1}{|\mathrm{Slot}|} \cdot \sum_{i=1}^{|\delta|} \delta_i$

function NegativeBenefitAvg($\mathrm{Slot}$)
    $\delta \leftarrow \varnothing$
    for $i \leftarrow 1$ to $n$ do
        $\lambda \leftarrow \mathrm{Benefit}(\mathrm{Slot}_i)$
        if $\lambda < 0$ then
            $\delta \leftarrow \delta \cup \{\lambda\}$
    return $\frac{1}{|\mathrm{Slot}|} \cdot \sum_{i=1}^{|\delta|} \delta_i$
```

It is important to note that when averaging the benefits, the number of values in the entire slot is used as the divisor, and not the cardinality of δ, because otherwise the positive benefit average function would not be monotonically increasing. This effect can be observed in Figure 4.9 and Figure 4.10.

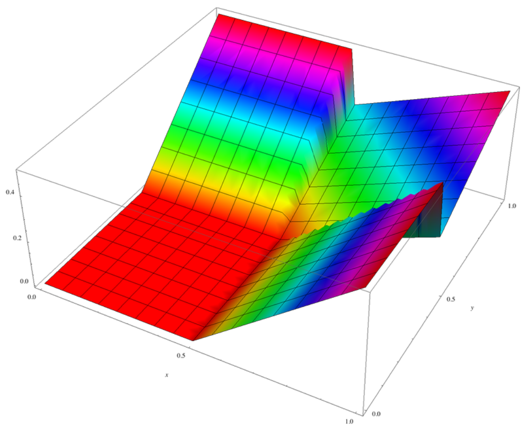

Figure 4.9: Plot of the positive benefit average function for two users using the δ cardinality when averaging (not monotonically increasing)

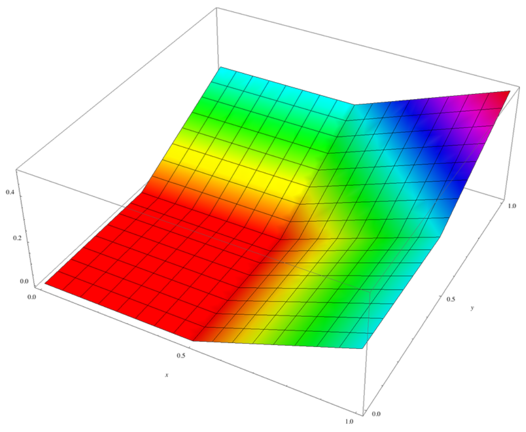

Figure 4.10: Plot of the positive benefit average function for two users using the Slot cardinality when averaging (monotonically increasing)

	Slot 1	Slot 2	Slot 3	Slot 4
User 1	1.0	0.0	0.1	0.3
User 2	0.7	0.95	0.05	0.5
User 3	0.1	0.0	0.2	0.6
Mean	0.6	0.31	0.12	0.46

Table 4.4: Satisfaction data with applied mean

Table 4.5 shows the $\mathrm{Benefit}$ function applied to each satisfaction value as well as the calculated positive and negative benefit averages, showing that while Slot 1 has a higher overall satisfaction, the desatisfaction is also greater than in Slot 4. Using this data, fine-grained decisions can be made based on the satisfaction and desatisfaction of meeting attendees, so that, depending on the strategy that the meeting initiator agent pursues, it can now decide whether or not it wants to accept the desatisfaction of some for the overall satisfaction of others.

Based on the benefit averages, two simple proposal selection strategies can then be employed:

Most satisfied users Maximize the number of satisfied users by picking a Slot with a high positive benefit average. A parameter for this strategy is the minimum

	Slot 1	Slot 2	Slot 3	Slot 4
User 1	0.5	-0.5	-0.4	-0.2
User 2	0.2	0.45	-0.45	0.0
User 3	-0.4	-0.5	-0.3	0.1
PBA	0.23	0.15	0.0	0.03
NBA	-0.13	-0.33	-0.38	-0.07

Table 4.5: Benefit data with positive and negative benefit average

acceptable negative benefit average, which controls how desatisfied the agent allows some users to be for the advantage of others.

Least unsatisfied users Minimize the number of unsatisfied users by sacrificing high satisfaction ratings for some users, i.e. pick the time slot with the negative benefit average closest to 0, and, if several such slots exists, pick the one with the highest positive benefit average.

4.5 Conclusion

In this chapter, a generic architecture for a meeting scheduling system has been devised, based on an agent communication layer specified by the Foundation of Intelligent Physical Agents (FIPA). The main design decision was the separation of passive personal agent and active meeting agent, which subsequently allowed a comprehensible scheduling protocol to be constructed. Lastly, a technique for best time slot selection was presented.

The following section will concern itself with the implementation and evaluation of a prototype based on the devised architecture. Said prototype will implement the scheduling protocol as well as time slot selection mechanism, demonstrating the feasibility of the components.

5 Implementation and Evaluation

This chapter addresses the implementation and evaluation of a multi-agent meeting scheduling system prototype implemented based on the requirements compiled in Chapter 3 (Analysis) as well as the design decisions made in Chapter 4 (Design).

The implementation is a proof-of-concept, intended to evaluate the system architecture and gain a better understanding of the inherent complexity that lies in designing and implementing a fully distributed agent based system capable of scheduling meetings, and to investigate the design's ability to accommodate different agent behavior implementations.

5.1 Environment

Scala 2.9 was chosen as the implementation language for the prototype. Given that Scala runs on the JVM[1], it is highly portable and able to use the large amount of freely available Java libraries. According to Martin Odersky, lead designer of Scala, it is an excellent basis for concurrent, parallel and distributed computing (Odersky, 2011).

The agent communication layer was implemented using "Akka", a library for building concurrent and distributed applications in Java and Scala using Actors[2].

5.2 Implementation approach

For the prototype, a behavior-driven development (BDD) technique was employed. Being derived from test-driven development with a focus on high-level behavioral details, BDD style development allowed for an easy translation of the specifications

[1] **Java Virtual Machine**: The code execution component of the Java software platform, which has been ported to many different environments.

[2] **Actor**: An actor is a computational entity which has its own thread of control, manages its own internal state and communicates with other actors by explicitly sending messages. (Lauterburg et al., 2009)

introduced in Section 4.3 (Meeting scheduling protocol) into a code based specification that the prototype can be tested against as the implementation progresses. The test framework used for this task was ScalaTest.

The test suite for the Meeting Agent serves as a good example for the developed tests, which are the following:

Listing 14: Behavioral tests for the Meeting Agent component

```
1    A meeting agent
2    - must process meeting requests
3    - must send out meeting proposals
4    - must no longer propose to participants who opted out of the meeting
5    - must request meeting proposal rating after timeout if not all invitees respond
6    - must send reservation request after a proposal has been accepted
```

The scenarios roughly relate to the three phases of scheduling process:

- Initiation phase (line 2 and 3)
- Meeting time slot finding phase (line 3-5)
- Meeting time slot reservation phase (line 6)

Behavior-driven test scenarios are usually written in a blackbox style, i.e. disregarding implementation details and simply testing the output given a series of inputs. The test scenario in Listing 15 serves to illustrate that technique. The `withTestSetup` method is invoked prior to the test in order to initialize the environment, including the Meeting Agent itself, the `Meeting` object and a list of example `MeetingTimeProposals`. The test then sends a `MeetingTimeProposals` message to the Meeting Agent and expects a single `MeetingTimeProposal` in response. Given that all three participants send the same proposal, the MA is expected to propose the proposal it just received.

BDD makes it easy to write down and test the designed behavior, and, if behavioral changes are necessary, backport those into the architectural behavior specification. Test-driven development in general is useful because it enables fast testing of implementation changes against the entire specification.

Listing 15: Example test taken from the Meeting Agent test suite

```scala
"A meeting agent" must {
  "send out meeting proposals" in withTestSetup {
    (ma, probe1, probe2, meeting, proposals) => {
      val mtps = MeetingTimeProposals(
        meeting,
        List(proposals.last)
      )

      val msg = Propose(
        Content("meeting-time-proposals", mtps) inReplyTo(meeting.uuid)
      )

      val expectedResponse = Propose(
        Content("meeting-time-proposal", proposals.last) inReplyTo(msg)
      )

      ma ! msg
      probe1.send(ma, msg)
      probe2.send(ma, msg)

      expectMsg(expectedResponse)
      probe1.expectMsg(expectedResponse)
      probe2.expectMsg(expectedResponse)
    }
  }
}
```

5.3 Important implementation details

For the prototype implementation, the calendar data and preferences are retrieved by using the iCalendar file format, that is, the `iCalendarConnector` class implements both `PreferencesConnector` and `CalendarConnector`. The preferences retrieval from the iCalendar format was implemented using a calendar extension with the name of `X-Preference`, i.e. the satisfaction values are hardcoded in the calendar. A more advanced `PreferencesConnector` implementation could feature, for example, mechanisms of preference estimation.

It is important to note that the agent communication and scheduling protocol implementation accounts for the largest part of the implementation work. The entire implementation, without tests, has about 2000 lines of code (LOC), of which only 600 LOC account for the agent behavior, that is, the `PreferencesConnector`, `CalendarConnector` and `CalendarAnalyzer` implementations of the personal agent system as well as the `DecisionMaker` and `TimeSlotSelector` implementations of the meeting agent. This shows that once a foundation implementation of the system has been written, one can focus on the important behavioral elements of the system without worrying about the underlying modules, given that those don't have to be modified in order to implement different scheduling algorithms and heuristics.

5.4 Evaluation

In order to evaluate the implementation, different meeting scheduling scenarios will be examined, analyzing the conversations between the agents that occur in the scheduling process.

5.4.1 Single participant scenario

The first scenario requires the meeting agent to find a free meeting time slot for a single person. This scenario serves as a good example of a simple conversation held between the personal agent and the meeting agent, and it verifies the basic scheduling functionality. The scenario is based on the calendar and preferences shown in Figure 5.1.

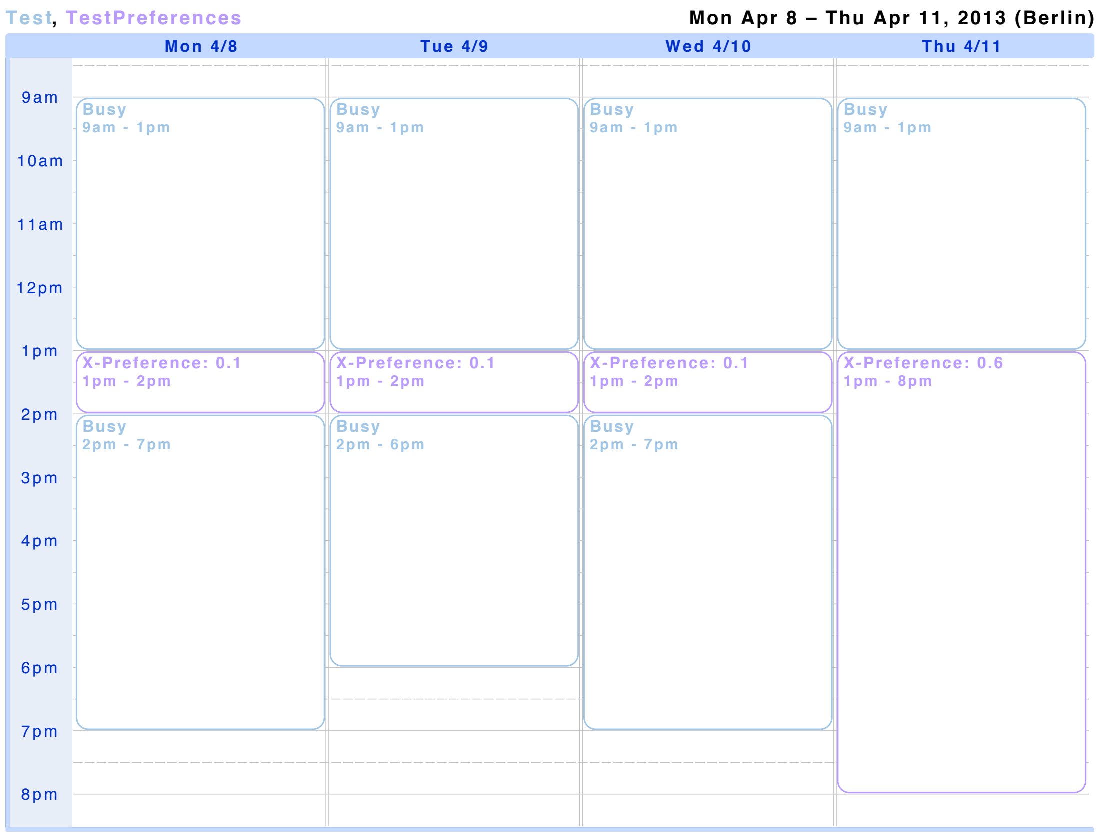

Figure 5.1: Four-day example calendar and preferences of a single person.

The message log for this scenario can be observed in Listing 16. Said log begins after the MA has broadcasted the meeting request, and thus the MA waits for proposals from the PA. The PA then sends a proposal for 2013–04–11 at 13:00, given that it is the earliest fitting time slot with the highest preference value. After receiving the proposal, the MA decides that it will propose the proposal it just received as a time for the meeting (which will obviously succeed), and then again enter the `WaitingForProposals` state. The PA accepts the message (line 22), and the MA then decides to try to reserve the proposal that was accepted. After receiving the `Confirm(Reservation)` messages in line 47, it sends a `Confirm(Reservation)` of its own, thus concluding the scheduling process, which the PA states in line 75.

Listing 16: Message log for the single participant scenario

```
 1 MA  :: Waiting for proposals.
 2 MA  :: Received proposals from PersonalAgent: MeetingTimeProposals(
 3         List(MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.6))
 4       )
 5 MA  :: Deciding next action because we received responses from all
 6       participants
 7 MA  :: Deciding next action through the DecisionMaker
 8 MA  :: Performing next action: Decision(
 9           WaitingForProposals(),
10           Some(
11               Propose(
12                   meeting-time-proposal, 44b90..,
13                   MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.6)
14               ), Id: 8
15           ), None
16       ) for participants: List(Agent(PersonalAgent))
17 MA  :: Broadcasting the following message to 1 recipients: Propose(
18           meeting-time-proposal, 44b90..,
19           MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.6),
20           Id: 8)
21 MA  :: Waiting for proposals.
22 MA  :: Received Accept(proposal) from PersonalAgent:
23           MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.6)
24 MA  :: Deciding next action because we received responses from all participants
25 MA  :: Deciding next action through the DecisionMaker
26 MA  :: Performing next action: Decision(
27           WaitingForReservationConfirmation(),
28           Some(
29               Request(
30                   reservation, 44b90..,
31                   Reservation(
32                       Meeting(44b90..),
33                       MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.0),
34                       2013-04-11T22:00:00.000Z
35                   ), Id: 11
36               )
```

```
            ), None
        ) for participants: List(Agent(PersonalAgent))
MA  :: Broadcasting the following message to 1 recipients: Request(
            reservation, 44b90..,
            Reservation(
                Meeting(44b90..),
                MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.0),
                2013-04-11T22:00:00.000Z
            ), Id: 11)
MA  :: Now waiting for reservation confirmations.
MA  :: Received reservation confirmation from sender PersonalAgent: Reservation(
            Meeting(44b90..),
            MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.0),
            2013-04-11T22:00:00.000Z
        )
MA  :: Deciding next action through the DecisionMaker
MA  :: Performing next action: Decision(
            WaitingForMeetingChanges(),
            Some(
                Confirm(
                    reservation, 44b90..,
                    Reservation(
                        Meeting(44b90..),
                        MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.0),
                        2013-04-11T22:00:00.000Z
                    ), Id: 14
                )
            ),None
        ) for participants: List(Agent(PersonalAgent))
MA  :: Broadcasting the following message to 1 recipients: Confirm(
            reservation, 44b90..,
            Reservation(
                Meeting(44b90..),
                MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.0),
                2013-04-11T22:00:00.000Z
            ), Id: 14
        )
MA  :: Now waiting for meeting changes
PA  :: Meeting Meeting(44b90..) successfully scheduled to
            MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.0)
```

As can be seen in the previous log, most of the log messages are recorded by the meeting agent. However, that is a minor implementational decision and has no further importance in the context of the scheduling process.

5.4.2 Four participants scenario

In order to evaluate the agents' behavior when scheduling a meeting with more than one participant, the four-day calendar shown in Figure 5.2 will be used. For this scenario, only one of the participants' agents has explicitly defined preferences (through the `iCalendarConector`). The other agents use a dummy implementation of the `PreferencesConnector` interface, which results in a satisfaction value of `0.5` for all meeting time slots in which no meeting has been scheduled yet.

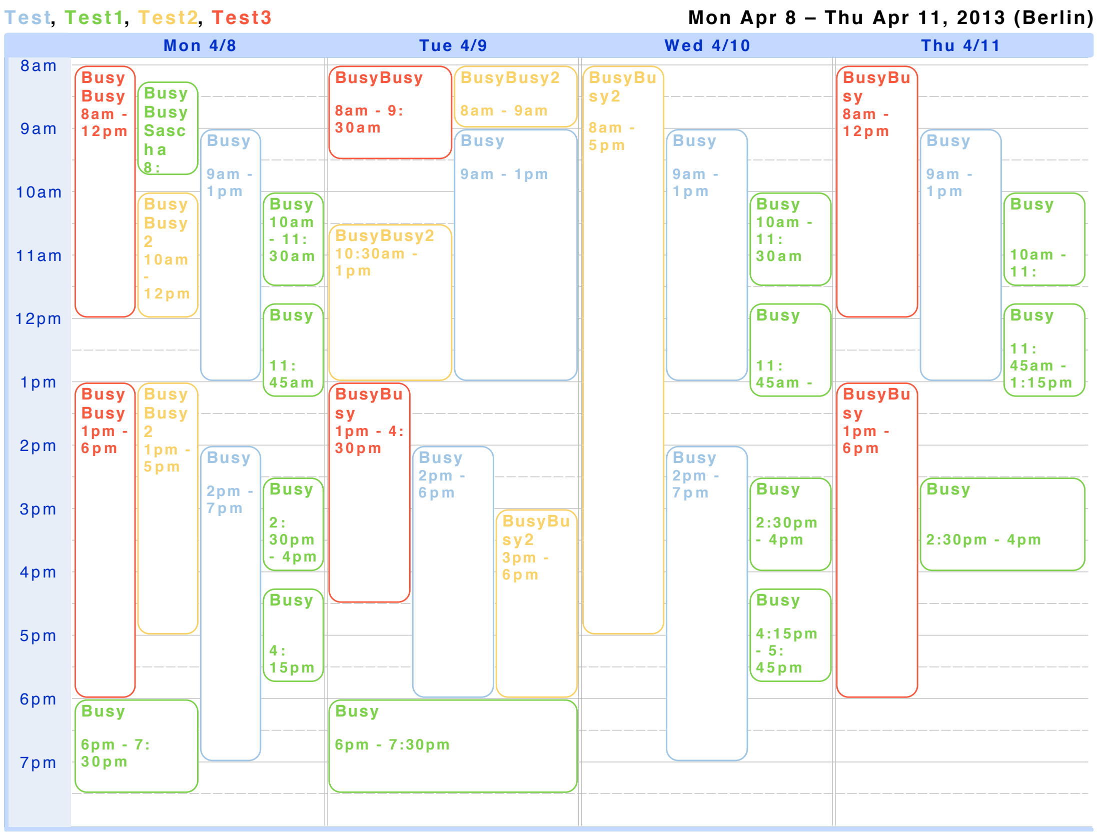

Figure 5.2: Four-day example calendar used for evaluation. The earliest time slot available for all participants is thursday at 18:00h

Given that the entire message log is too large to include, selected messages from the process, that give insight into the scheduling behavior, will be shown. In this scenario, the initial proposals made by the four PAs have no intersection, and therefore the meeting agent must inquire more information about their calendars. In the prototype implementation, the MA tries to find a meeting time slot by simply asking the PAs for proposals until a consensus is found. While this is not the most efficient technique, it demonstrates that the devised scheduling protocol is capable of fulfilling its purpose. More complex heuristics may then be implemented in the MA. The following log messages represent the initial phase of the scheduling process:

Listing 17: Initial messages from the four participants scenario

```
MeetingAgent  :: Waiting for proposals.
MeetingAgent  :: Received proposals from sender PersonalAgent2: MeetingTimeProposals(
    List(MeetingProposal(2013-04-08T12:00:00.000+02:00, 0.5))
)
MeetingAgent  :: Received proposals from sender PersonalAgent3: MeetingTimeProposals(
    List(MeetingProposal(2013-04-08T08:00:00.000+02:00, 0.5))
)
MeetingAgent  :: Received proposals from sender PersonalAgent: MeetingTimeProposals(
    List(MeetingProposal(2013-04-08T20:00:00.000+02:00, 0.5))
)
MeetingAgent  :: Received proposals from sender PersonalAgent: MeetingTimeProposals(
    List(MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.6))
)
```

Given that the proposals have no intersection, the MA now inquires ratings for all four proposals, beginning with the one with the highest ratings, in order to see if any of them is accepted by all four participants. The following log shows the rating inquiry for the first proposal.

Listing 18: Inquiry of meeting time proposal ratings by the Meeting Agent

```
MeetingAgent  :: Performing next action: Decision(
    WaitingForRatings(3),
    None,
    Some(
        Map(
            PersonalAgent2 -> Request(
                meeting-time-proposal-rating, b6b27..,
                MeetingTimeProposalRating(
                    MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.0)
                ), Id: 11),
            [...]
        )
    )
)
MeetingAgent  :: Now waiting for proposal ratings.
MeetingAgent  :: Received meeting time proposal ratings from sender PersonalAgent2:
    MeetingTimeProposalRating(
        MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.0)
    )
```

```
MeetingAgent  :: Received meeting time proposal ratings from sender PersonalAgent3:
    MeetingTimeProposalRating(
        MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.5)
    )
MeetingAgent  :: Received meeting time proposal ratings from sender PersonalAgent4:
    MeetingTimeProposalRating(
        MeetingProposal(2013-04-11T13:00:00.000+02:00, 0.0)
    )
```

As can be seen in the log, two of the PAs rate the meeting time proposal with a satisfaction value of `0.0`, meaning that they have already a meeting scheduled at the time. As a result, the MA continues to inquire additional proposals and ratings, until a consensus is found. In the example, this consensus is 2013–04–11 at 18:00h. The last rounds of negotiation for this scenario can be observed in Listing 19. PersonalAgent1 proposes the soon-to-be final time slot in line 2, which is then proposed to all other agents, because it is the highest rated time slot available. Given that all PAs are free at the suggested time, it is accepted by all (line 16–23) and subsequently reserved by the MA (line 26–49). Finally, the MA confirms the meeting in line 52 whereupon the PAs report the successfully scheduled proposal in line 61–71.

Listing 19: Conclusion of the four participant scenario

```
 1 MeetingAgent  :: Now waiting for proposal ratings.
 2 MeetingAgent  :: Received meeting time proposal ratings from sender PersonalAgent1:
 3     MeetingTimeProposalRating(MeetingProposal(2013-04-11T18:00:00.000+02:00, 0.6))
 4 MeetingAgent  :: Deciding next action through the DecisionMaker
 5 MeetingAgent  :: Performing next action: Decision(
 6     WaitingForProposals(),
 7     Some(
 8         Propose(
 9             meeting-time-proposal, b6b27..,
10             MeetingProposal(2013-04-11T18:00:00.000+02:00, 0.6), Id: 458)
11         ),None
12     )
13 )
14 MeetingAgent  :: Broadcasting the following message to 4 recipients: Propose(meeting-time-proposa
15 MeetingAgent  :: Waiting for proposals.
16 MeetingAgent  :: Received Accept(proposal) from sender PersonalAgent2:
17     MeetingProposal(2013-04-11T18:00:00.000+02:00, 0.5)
18 MeetingAgent  :: Received Accept(proposal) from sender PersonalAgent3:
19     MeetingProposal(2013-04-11T18:00:00.000+02:00, 0.5)
20 MeetingAgent  :: Received Accept(proposal) from sender PersonalAgent4:
21     MeetingProposal(2013-04-11T18:00:00.000+02:00, 0.5)
22 MeetingAgent  :: Received Accept(proposal) from sender PersonalAgent1:
23     MeetingProposal(2013-04-11T18:00:00.000+02:00, 0.6)
24 MeetingAgent  :: Deciding next action because we received responses from all participants
```

```
MeetingAgent  :: Deciding next action through the DecisionMaker
MeetingAgent  :: Performing next action: Decision(
    WaitingForReservationConfirmation(), ...
)
MeetingAgent  :: Broadcasting the following message to 4 recipients: Request(
    reservation,
    b6b27..,
    Reservation(
        Meeting(b6b27..),
        MeetingProposal(2013-04-11T18:00:00.000+02:00, 0.0),
        2013-04-11T22:00:00.000Z
    ), Id: 464
)
MeetingAgent  :: Now waiting for reservation confirmations.
MeetingAgent  :: Received reservation confirmation from sender PersonalAgent2:
    Reservation(...)
MeetingAgent  :: Now waiting for reservation confirmations.
MeetingAgent  :: Received reservation confirmation from sender PersonalAgent3:
    Reservation(...)
MeetingAgent  :: Now waiting for reservation confirmations.
MeetingAgent  :: Received reservation confirmation from sender PersonalAgent4:
    Reservation(...)
MeetingAgent  :: Now waiting for reservation confirmations.
MeetingAgent  :: Received reservation confirmation from sender PersonalAgent1:
    Reservation(...)
MeetingAgent  :: Deciding next action through the DecisionMaker
MeetingAgent  :: Performing next action: Decision(WaitingForMeetingChanges(), ...)
MeetingAgent  :: Broadcasting the following message to 4 recipients: Confirm(
    reservation,
    b6b27..,
    Reservation(Meeting(b6b27..),
        MeetingProposal(2013-04-11T18:00:00.000+02:00, 0.0),
        2013-04-11T22:00:00.000Z
    ), Id: 470
)
MeetingAgent  :: Now waiting for meeting changes
PersonalAgent :: Meeting Meeting(b6b27..) successfully scheduled to MeetingProposal(
    2013-04-11T18:00:00.000+02:00, 0.0
)
PersonalAgent :: Meeting Meeting(b6b27..) successfully scheduled to MeetingProposal(
    2013-04-11T18:00:00.000+02:00, 0.0
)
PersonalAgent :: Meeting Meeting(b6b27..) successfully scheduled to MeetingProposal(
    2013-04-11T18:00:00.000+02:00, 0.0
)
PersonalAgent :: Meeting Meeting(b6b27..) successfully scheduled to MeetingProposal(
    2013-04-11T18:00:00.000+02:00, 0.0
)
```

5.4.3 Four participants scenario - variations

If the meeting were to be scheduled in the first three days of the week depicted in Figure 5.2 (in contrast to four days that were used in the previous section), a meeting time consensus could not be found, because the first meeting time slot that is available for all PAs is on the fourth day. Given that the prototype implementation does not support conflict resolution, this results in the MA reporting a failed meeting scheduling process:

Listing 20: Scheduling failure variation of the four participant scenario

```
MeetingAgent  :: Waiting for proposals.
PersonalAgent :: No further meeting time proposals for Meeting(ad35a..).
MeetingAgent  :: Received failure from sender PersonalAgent1:
    Failure(Meeting(ad35a..))
MeetingAgent  :: Deciding next action through the DecisionMaker
MeetingAgent  :: Performing next action: Decision(AbortScheduling(), ...)
MeetingAgent  :: Broadcasting the following message to 4 recipients: Failure(
    meeting,
    ad35a..,
    Meeting(ad35a..),
    Id: 51
)
```

In order to provide an example of the designs flexibility, a slightly different example of the `DecisionMaker` will be used, which simply discards meeting participants that report a meeting scheduling failure such as the one previously shown. The other modules of the MA, that is the `TimeSlotSelector` and the `SchedulingControl`, were not modified for this task. Using the modified `DecisionMaker` results in the following log excerpt:

Listing 21: Discarding conflicting participant in four participant scenario

```
MeetingAgent  :: Waiting for proposals.
PersonalAgent :: No further meeting time proposals for Meeting(2fdc3..).
MeetingAgent  :: Received failure from sender PersonalAgent1:
    Failure(Meeting(2fdc3..))
MeetingAgent  :: Deciding next action through the DecisionMaker
MeetingAgent  :: Removing PersonalAgent1 from the list of participants
MeetingAgent  :: Performing next action: Decision(
    WaitingForProposals(),
    Some(
        Propose(
            meeting-time-proposal, 2fdc3..,
            MeetingProposal(2013-04-08T20:00:00.000+02:00, 0.5), Id: 51)
        )
    ),
    None
)
MeetingAgent  :: Broadcasting the following message to 3 recipients:
    Propose(
        meeting-time-proposal,
        2fdc3..,
        MeetingProposal(2013-04-08T20:00:00.000+02:00, 0.6),
        Id: 51
    )
```

In this log excerpt, instead of aborting the meeting scheduling process when PersonalAgent1 reports a failure, the agent is simply removed from the list of participants. After removing the agent, the MA then continues with a new proposal for the remaining agents. Such a modification of the MA's behavior by means of the `DecisionMaker` component is an example of how, without modifying the core communication functionality of a meeting agent (located in the `SchedulingControl` module), its behavior can be easily modified.

5.5 Conclusion

In this chapter, a prototype implementation of the previously devised architecture for a meeting scheduling system based on the multi-agent paradigm has been devised and evaluated. In order to provide some context for the evaluation, the implementation environment and approach was described. Subsequently, the prototype was evaluated by means of two example scenarios:

- The single participant scenario was used to provide a clear overview of the conversation between a meeting agent and a personal agent. Such an overview is difficult to achieve with a larger scenario, given that the amount of messages generated increases dramatically.
- A scenario with four participants scheduling a meeting on a four-day calendar. This scenario served to provide a view into the negotiation process that is carried out, and how it can be influenced by changing the implementations of some of the meeting agents' modules.

The evaluation shows that the prototype implementation based on the devised system architecture is capable of autonomously scheduling meetings, and that it is possible to exchange the agents' scheduling behavior by *selectively* modifying agent modules.

6 Conclusion

The purpose of this thesis was to provide a flexible multi-agent architecture for an automated meeting scheduling system. To achieve this goal, the state of the art in multi-agent based meeting scheduling had to be collected in the literature review. Subsequently, the gained knowledge was then molded into a scenario which laid the foundation for the requirements analysis. In the design chapter, an architecture for a system capable of automated meeting scheduling based on the notion of multi-agents was then devised along with a generic meeting scheduling communication protocol. The focus was laid on the architectures ability to accommodate different behaviors and data evaluation methods. Lastly, a proof-of-concept implementation for said architecture was developed and evaluated based on the previously established requirements, demonstrating the design's feasibility.

In summary it can be said that the goals of this thesis have been reached by providing a flexible architecture for future meeting scheduling systems and proving that the resultant implementation is capable of housing different behavioral elements without requiring excessive implementational effort.

Ultimately, another step was taken in the direction of automating a specific tedious every day task in the professional and personal life of many people, with the greater goal of reaching a state where users no longer think of such computer systems as mere tools, but as their "technological companions" which adapt themselves *reliably* to their users' abilities, preferences and requirements.

Outlook

Though the basic mechanisms in meeting scheduling are relatively easily captured, the more intricate and implicit processes taking place when someone makes a decision about whether or not to schedule a meeting at a given time are more difficult to grasp.

As an example, in this thesis, the user's preferences have been regarded as a very abstract piece of information, some numerical value, even though it can be considered one of, if not *the* most vital information when scheduling a meeting. A few approaches for preferences retrieval have been mentioned, including machine learning and aided user input, but the possibilites for enhancements in this area are enormous. For instance, let's take the location of two given meetings X and Y. If these two meetings are scheduled close to one another, they should preferably take place at the same location, and if not, the time gap between the two meetings must allow the participant to travel from meeting X's location to meeting Y's location. Additionally, the scheduling system must ensure that no other meeting is scheduled in the time gap that is reserved for travel. To increase the complexity, one could also assume that a meeting must be scheduled which requires the participant to travel by plane and which takes about three days. One could also assume that a return flight after three days is many times as costly as a return flight after seven days, and a system aware of that fact could book a return flight after seven days and schedule a number of networking appointments in the remaining in order to increase efficiency.

The previous example might seem overly specific, yet such complex evaluations are very important if a system is really to be accepted as a replacement for a job as important as a personal secretary, rather than as a tool that aids the scheduling processes, which leads to several questions that remain to be answered, such as:

- Is it generally possible to develop a system that is able to perform complex tasks like scheduling meetings autonomously without constant supervision of the results?
- And if so, would people be willing to completely delegate a task as important as the organization of their weekly schedule to a computer system?

In the narrower context of this thesis there are also several questions that remain unanswered as well as features that remain to be devised, which would serve as excellent starting points for further research and development, in addition to the vast field of preference estimation:

- How can efficient conflict resultion strategies be incorporated into the presented meeting scheduling system design and communication protocol?

- How can scheduling fairness be obtained if the assumptions that personal agents give truthful answers for their satisfaction values is discarded?
- How can meetings be re-scheduled if that would improve the overall satisfaction for the personal agent's user, and how can the cost of re-scheduling a meeting be weighted against the benefit the arises from such a re-scheduling process?
- How could a "final approval" mechanism be provided to the user? That is, a mechanism that asks the user for final confirmation after the automated meeting scheduling process has completed. This is an important question, because such a mechanism could dramatically increase the acceptance of the system and at the same time provide information for machine learning mechanisms in order to improve the next scheduling process. Said mechanism would have to present all information of and surrounding the scheduled meeting to the user in a clearly laid out way, for example on the user's smartphone device in order to reach him more quickly than on a desktop computer.
- It is not currently possible for someone to join a meeting on one's own initiative or to be invited by a participant of the meeting which is not the meeting initiator.
- The negotiation process devised in this thesis is one dimensional, whereas human negotiation processes are almost always multidimensional. For example, someone might put up with an unsatisfactory meeting time, considering that the last time the meeting was held, the time was very satisfactory for him while being unsatisfactory for the other participants (global optimum instead of only local considerations). Therefore, an important question is: How can multidimensional scheduling techniques be integrated into the meeting scheduling process in general, and into the devised system in particular?

In conclusion, it can be said that the presented meeting scheduling system architecture can serve as a solid foundation for further research, which is urgently needed in this research field.

Bibliography

Biswas, J. and Bhonsle, S. (1992). Distributed scheduling of meetings: a case study in prototyping distributed applications, *Proceedings of the Second International Conference on Systems Integration* pp. 656–665.
URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=217265

Black, A. P. (1986). The Eden Project: Overview and Experiences, *Autumn'86 Conference Proceedings: Manchester, UK*, EUUG (Society). Autumn Conference and European UNIX Systems User Group. Autumn Conference, Manchester, UK, pp. 177–189.
URL: http://books.google.de/books?id=iFDmNQAACAAJ

Bradner, S. (1997). Key words for use in RFCs to Indicate Requirement Levels.
URL: http://www.ietf.org/rfc/rfc2119.txt (last accessed: 2012-05-05)

Bui, H., Venkatesh, S. and Kieronska, D. (1995). A multi-agent incremental negotiation scheme for meetings scheduling, *Proceedings of Third Australian and New Zealand Conference on Intelligent Information Systems. ANZIIS-95* pp. 175–180.
URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=705736

Chun, A., Wai, H. and Wong, R. Y. (2003). Optimizing agent-based meeting scheduling through preference estimation, *Engineering Applications of Artificial Intelligence* **16**(7-8): 727–743. ISSN: 09521976.
URL: http://linkinghub.elsevier.com/retrieve/pii/S0952197603001167

Crawford, E. and Blum, M. (2009). *Learning to Improve Negotiation in Semi-Cooperative Agreement Problems Manuela Veloso , Chair*, Doctoral thesis, Carnegie Mellon University, Pittsburgh, PA.
URL: http://reports-archive.adm.cs.cmu.edu/anon/anon/usr0/ftp/2009/CMU-CS-09-111.pdf

Crawford, E. and Veloso, M. (2005a). Learning Dynamic Preferences in Multi-Agent Meeting Scheduling, *Intelligent Agent Technology, IEEE/WIC/ACM International Conference on* pp. 3–6.
URL: http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=1565590

Crawford, E. and Veloso, M. (2005b). Learning Dynamic Time Preferences in Multi-Agent Meeting Scheduling Elisabeth Crawford, *Technical Report July*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.
URL: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA457066

Crawford, E. and Veloso, M. (2005c). Learning to select negotiation strategies in multi-agent meeting scheduling, *Progress in Artificial Intelligence* **Volume 380**: 584–595.
URL: http://www.springerlink.com/index/7210140k27370611.pdf

Foundation for Intelligent Physical Agents (2002a). FIPA Abstract Architecture Specification - SC00001L.
URL: http://fipa.org/specs/fipa00001/SC00001L.pdf

Foundation for Intelligent Physical Agents (2002b). FIPA ACL Message Structure Specification - SC00061G.
URL: http://www.fipa.org/specs/fipa00061/SC00061G.pdf

Foundation for Intelligent Physical Agents (2002c). FIPA Communicative Act Library Specification - SC00037J.
URL: http://www.fipa.org/specs/fipa00037/index.html

Holman, C. and Almes, G. (1985). The Eden Shared Calendar System, TR 85-05-02, *Technical Report*, University of Washington, Seattle.

Jennings, N. R., Sycara, K. and Wooldridge, M. (1998). A Roadmap of Agent Research and Development, *Autonomous Agents and Multi-Agent Systems* **1**(1): 7–38. ISSN: 1387-2532.
URL: http://www.springerlink.com/content/1387-2532/1/1/

Johansen, D. and Anshus, O. (1988). A Distributed Diary Application, *in* A. Cerveira (ed.), *Computer Communication Systems*, Elsevier Science Publishers, Amsterdam, pp. 77–82.

Kincaid, C. M., Dupont, P. B. and Kaye, A. R. (1985). Electronic Calendars in the Office : An Assessment of User Needs and Current Technology, *ACM Transactions on Information Systems* **3**(1): 89–102. ISSN: 1046-8188.
URL: http://www.sis.pitt.edu/~dist/coursePages/IS2470/p89kincaid.pdf

Lai, T. E. N., Selamat, M. H. and Muda, Z. (2009). IMROVING AGENT-BASED MEETING SCHEDULING, *Journal of Theoretical and Applied Information Technology* **6**(2): 156–164.
URL: http://www.jatit.org/volumes/research-papers/Vol6No2/3Vol6No2.pdfhttp://www.jatit.org/volumes/sixth_volume_2_2009.php

Lauterburg, S., Dotta, M., Marinov, D. and Agha, G. (2009). A Framework for State-Space Exploration of Java-Based Actor Programs, *2009 IEEE/ACM International Conference on Automated Software Engineering* pp. 468–479.
URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5431748

Mattern, F. and Sturm, P. (1989). Automatic distributed calendar and appointment system, *Microprocessing and Microprogramming* **27**(1): 455–462. ISSN: 01656074.
URL: http://dx.doi.org/10.1016/0165-6074(89)90091-4;

Minsky, M. L. (1988). *The society of mind*, Simon & Schuster Inc., New York. ISBN: 0-671-60740-5.
URL: http://www.amazon.com/Society-Mind-Marvin-Minsky/dp/0671657135

Mintzberg, H. (1973). *The Nature of Managerial Work*, Longman, New York. ISBN: 978-0060445560.
URL: http://www.amazon.de/Nature-Managerial-Work-Henry-Mintzberg/dp/0060445564

Odersky, M. (2011). Why Scala?
URL: http://blog.typesafe.com/why-scala (last accessed: 2012-08-28)

Oh, J. and Smith, S. F. (2004). Learning calendar scheduling preferences in hierarchical organizations, *Proceedings of 6th International Workshop on Preferences and Soft Constraints (CP'04)* .
URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.84.6462&amp;rep=rep1&amp;type=pdf

Romano, N. C. and Nunamaker, J. F. (2001). Meeting Analysis : Findings from Research and Practice, *Proceedings of the 34th Hawaii International Conference on System Sciences*, Vol. 00, pp. 1–13.
URL: http://www.okstate.edu/ceat/msetm/courses/etm5221/Week1Challenges/MeetingAnalysisFindingsfromResearchandPractice.pdf

SFB Transregio 62 (2012).
URL: http://www.sfb-trr-62.de/ (last accessed: 2012-12-16 16:10)

Sugihara, K., Kikuno, T. and Yoshida, N. (1989). Meeting Scheduler for Office Automation, *IEEE Transactions on Software Engineering* **15**(10): 1141–1146.
URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=559760

Sugumaran, M. (2011). A Design of Centralized Meeting Scheduler with Distance Metrics, *International Journal of Computer Science and Information Technologies* **2**(3): 1001–1006. ISSN: 0975-9646.
URL: http://ijcsit.com/docs/Volume2/vol2issue3/ijcsit2011020313.pdf

Sugumaran, M., Easwarakumar, K. S. and Narayanasamy, P. (2003). A New Approach for Meeting Scheduler using A * -Algorithm, *TENCON 2003. Conference on Convergent Technologies for Asia-Pacific Region* **1**.
URL: http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=1273357

Teger, S. L. (1983). Factors Impacting the Evolution of Office Automation, *Proceedings of the IEEE* **71**(4): 503–511.

Glossary

Actor
An actor is a computational entity which has its own thread of control, manages its own internal state and communicates with other actors by explicitly sending messages. (Lauterburg et al., 2009). 49

Java Virtual Machine
The code execution component of the Java software platform, which has been ported to many different environments.. 49

Universally Unique Identifier
An identifier that has a unique value within some defined universe.. 38

List of Figures

List of listings

List of Tables

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 27.12.2012	
Ort, Datum	Unterschrift