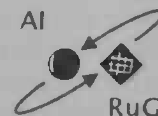
957
2006
006

Qualitative Decision Theory and Graph Rewriting in an Adaptive Diary Assistant

Joris IJsselmuiden*
studentnr. 1267523
August 2006

supervisors:
Rineke Verbrugge*
Tim Samshuijzen+
referee:
Bart Verheij*

* Artificial Intelligence, University of Groningen, Zernikepark 10, 9747 AN, Groningen, The Netherlands. Email: [ijssel, l.c.verbrugge, b.verheij]@ai.rug.nl



+ Rockingstone, Gen. Foulkesweg 30a, 6703 BS, Wageningen, The Netherlands. Email: tim@rockingstone.nl

Rockingstone

Kunstmatige Intelligentie
Rijks*universiteit* Groningen

Rijksuniversiteit Groningen
BIBLIOTHEEK
SOCIALE WETENSCHAPPEN

Abstract

Developing a multiagent system (MAS) is difficult due to numerous issues. We aim to define and discuss some of these. The project contains most of the typical phases of MAS development, from literature study to implementation (testing is beyond the scope of the project). As a case-study for the development of a multiagent system, we use an adaptive diary assistant that is run by a team of autonomous agents. Creating a useful application is only a secondary goal. It is the entire process from literature study to implementation that we are interested in, not the end product.

The two main techniques used for the development of this diary assistant are qualitative decision theory and graph rewriting. Qualitative decision theory provides us with a convenient architecture for the design, whereas graph rewriting is used for implementation. Besides defining and discussing the issues encountered during MAS development, we also explore and expand the possibilities of graph rewriting as a programming language for MAS. Functionality for qualitative decision theory and complex data manipulation is added to the graphical agent programming environment OutOfBrain.

The project has many goals, but the focus lies with two simple research questions: "How can Boutilier's QDT architecture be extended to allow for temporal reasoning?" and "Can Boutilier's QDT architecture, extended with a temporal modality, be integrated into OutOfBrain?".

QDT can possibly be extended with temporal reasoning by combining it with BDI's apparatus for handling time series and events. We call this combined architecture QDT+ from now on. It has three modalities, one for linear preference orderings, one for linear normality orderings and one for time trees that branch into the future and are linear in the past. This allows agents to reason about time as well as mental attitudes. We claim that such a multimodal architecture is necessary for general purpose learning. The design we end up with leads to the following conclusion. QDT can be extended to allow for temporal reasoning by combining it with certain parts of BDI. We provide the first steps toward such an architecture.

The second main goal of this project is to implement QDT+ as an integral part of OutOfBrain. In this endeavour we were only partly successful. The original QDT architecture, using preferences and normalities, was indeed implemented and it is now part of OutOfBrain. However, the functionality for handling time and events was not. This is partly due to time constraints on the project. Despite this deficit, we are tempted to answer our second research question in a positive way: QDT+ can indeed be implemented to become an integrated part of OutOfBrain. Although we did not succeed in doing so ourselves, we developed a clear picture of how one would go about implementing such an architecture.

The project has four secondary goals. They should be viewed as the context for our two main goals. Also, we use them to solidify the more abstract challenges of analysing issues in MAS and exploring and extending OutOfBrain. The first of these secondary goals is to create a useful application using QDT and OutOfBrain. Requirements on new software are extremely high nowadays and the adaptive diary assistant does not live up to them. Security issues for example are not dealt with. One's diary should be private and not accessible to other people. Achieving this is beyond the scope of this project. Besides this issue, many others have to be dealt with before the application could be called seaworthy. However, we decided that the diary assistant should never leave the computer-lab. It served its purpose, which was to be a test-bed to explore and extend QDT and OutOfBrain.

Secondary goal number two is to successfully incorporate agent communication in order to improve the application's performance and to further our understanding of sociality

in MAS. The communication standard we use, FIPA-ACL, proved to be highly intuitive and perfectly capable of modelling fairly complex communication flows. Therefore, we encourage other researchers and engineers to make use of it. The third context-goal is to develop a powerful learning method to facilitate user modelling. Again, we aim to improve the diary assistant with that, but we also desire to expand our own knowledge of the topic. We were only partly successful in this endeavour. The learning technique used in the application is not general purpose; it can only operate in a highly constrained, predefined problem space that fails to impress. The fourth and final secondary goal is to use standards from cognitive ergonomics to improve the interface of the application. We hardly pursued this goal in practice, but we did end up with a usable interface thanks to our own insights and Rockingstone's experience with developing database applications.

Because of our research goals, we choose to make use of a team of agents as opposed to a single agent. From an engineering point of view, it would be better to use a single agent, because the system is not distributed in space and henceforth not a true multiagent system. In the current version of the programming environment, the world, including the agents, is represented in a single graph. We simulate distributedness by posing constraints on the agents' scope of access and control over the environment. They need to work together, since each member has unique capabilities.

To successfully implement the BDI components as well, one would have to develop a way to store the history of the world. This would result in not one, but a whole series of OutOfBrain graphs, one for each time step. Each one is a world that can contain anything from first order sentences to complex statements, combining temporal operators with mental attitudes. This is something for potential future work.

Another possible topic for future studies is the general purpose learning method we mentioned. It would require a module that extracts logical formulas from raw input, coming from the real world, an experimental setting, a simulation or a database. Needless to say, this is an unsolved problem and many people are working on it. A learning method for QDT+ would have to transform histories of percepts to formulas containing normalities and preferences, combined with temporal operators.

We believe that MAS will play a central role in the future world of diary-keeping. The best approach is to have a single agent represent his user instead of having an entire team of agents per user. We envision a company where everybody caries a PDA that acts as a personal secretary. After a single button-press by the chairman of the board, a meeting can be scheduled that fits everybody's appointments and preferences as much as possible. The personal secretaries negotiate and form alliances in order to satisfy (or rather satisfice) the desires of their masters.

Table of contents

Acknowledgements ... 7
Preface ... 8
Legal notice ... 8
List of Figures ... 9
List of Tables ... 9

1 Introduction ... 10
1.1 Project motivations ... 10
1.2 Project goals ... 12
1.3 Applied techniques ... 13
1.3.1 Qualitative decision theory ... 14
1.3.2 Graph rewriting and OutOfBrain ... 14
1.3.3 Agent communication ... 15
1.3.4 User modelling ... 15
1.3.5 Interface design ... 16

2 Literature on qualitative decision theory ... 17
2.1 Some history on decision theory ... 18
2.2 Motivations for using qualitative methods ... 19
2.3 QDT logic ... 20
2.3.1 Introduction to QDT ... 20
2.3.2 The logic CO ... 22
2.3.3 The logic QDT ... 24
2.3.4 Properties of QDT ... 25
2.3.5 Knowledge based systems (KBS) ... 28
2.4 BDI logic ... 29
2.4.1 Simple BDI example ... 30
2.4.2 Formal definitions of BDI's logic and model ... 31
2.4.3 The BOID architecture ... 33
2.4.4 Collective intentions ... 35
2.5 Combining QDT and BDI ... 35

3 Literature on graph rewriting and other topics ... 38
3.1 Graph rewriting and OutOfBrain ... 38
3.1.1 Introduction to graph rewriting ... 39
3.1.2 Variations in graph rewriting mechanisms ... 39
3.1.3 Choosing a rewrite mechanism ... 42
3.1.4 Coloured Petri nets ... 43
3.1.6 Introduction to the OutOfBrain system ... 46
3.1.7 Relating OutOfBrain to other graph rewriting systems and Petri nets ... 48
3.3 User modelling ... 50
3.4 Interface design ... 51
3.4.1 Control components ... 51
3.4.2 Display components ... 51

4 Design ... 52
4.1 Introduction to QDT+ ... 52
4.2 Applying the QDT+ model to a diary assistant ... 53

4.2.1 The agents and their environment ... 53
4.2.2 Reasoning about the past ... 54
4.2.3 Cooperation strategies ... 56
4.3 Applying the QDT+ logic to a worked example ... 56
4.3.1 Example: the umbrella problem ... 57
4.3.2 Possible worlds and their expected utility ... 57
4.3.3 Summation over possible outcomes ... 59
4.3.4 Rule priority ... 60
4.3.5 Some thoughts on qualitativity and complex formulas ... 61
4.4 Designing the adaptive diary assistant ... 61
4.4.1 Preview of the diary assistant's functionality ... 62
4.4.2 Overview of methods ... 64
4.4.3 Overview of data ... 66
4.5 Methods for communication in a diary assistant ... 69
4.5.1 Communication during planning ... 70
4.5.2 Communication during distance-matrix maintenance ... 71
4.6 Methods for learning in a diary assistant ... 72
4.7 Interface design ... 74

5 Implementation ... 75
5.1 Design reconsideration ... 75
5.2 Process overview ... 77
5.3 Some rewrite rules explained ... 83
5.4 Using preferences ... 87
5.5 Test report ... 89

6 Discussion and conclusions ... 93
6.1 Evaluation ... 93
6.1.1 Goal evaluation ... 93
6.1.2 Results ... 94
6.1.3 Reasons for design reconsideration ... 96
6.2 Conclusions ... 97
6.3 Future work ... 99
6.3.1 Improvements to QDT+ ... 100
6.3.2 Improvements to OutOfBrain ... 101
6.3.3 A network of personal digital assistants (PDA's) ... 102

References ... 103
Appendix A: OutOfBrain manual ... 106
Appendix B: Overview of todo-agent and user-agent ... 153
Appendix C: host graph ... 156

Acknowledgements

I would like to thank Rineke Verbrugge for her excellent supervision. During our frequent meetings, we had some very inspiring discussions and I received an abundance of valuable comments on my thesis from her. Furthermore, my thanks go out to Tim Samshuijzen, the man behind OutOfBrain. It was a privilege to be part of the development of this new programming environment. Also, I would like to thank him for the many brainstorm sessions we had, in which theory and practice came together quite well. The entire crew at Rockingstone also deserves my thanks. I had a wonderful time in Wageningen and I will not forget you. Also, thanks to Bart Verheij for being the referee of this thesis. Finally, thanks to Mieke IJsselmuiden and Eva Jorritsma for their care and support.

Preface

This project is a joint effort by Artificial Intelligence at the University of Groningen and Rockingstone Robotics B.V. It serves as a graduation project for the Master Artificial Intelligence. Furthermore, it is a trial for OutOfBrain, the visual programming environment currently under development at Rockingstone.

The Artificial Intelligence department of the University of Groningen is known for its diversity and interdisciplinary character. The four main streams of science practiced there are autonomous and perceptive systems, multiagent systems, cognitive modelling and language, sound and cognition. The department has two main goals. One is to study cognition in all its versatility. The other is to engineer useful, intelligent devices. History tells us that a healthy cross-pollination is possible between these two goals. The department will continue to strive for this goal in the future.

Rockingstone is a company that consists of three branches: Rockingstone IT, Rockingstone Trading and Rockingstone Robotics. The IT branch specializes in database applications. Their flagship is ILAB, a database system for antique books that is used all over the world. Rockingstone Trading is a wholesale dealer in robotic toys and gadget. Since these two branches are quite profitable, the robotics branch can afford to have an long term, innovative strategy. OutOfBrain is the first product emerging from Rockingstone Robotics.

There are four people involved in the project. The general task allocation can be described as follows. Joris IJsselmuiden is responsible for the theoretical aspects of development, whereas Tim Samshuijzen largely takes care of the practical complement thereof. In reality, the border between theory and practice is not that strict. Joris takes care of a substantial part of the implementation and Tim performed many theoretical tasks as well. Rineke Verbrugge's tasks consist mostly of supervision and guidance. Finally, Bart Verheij is the referee of this Master's thesis.

The literature study required for this project, is carried out by Joris IJsselmuiden, under the guidance of Rineke Verbrugge. The design of OutOfBrain's qualitative decision module as well as the design of the adaptive diary assistant are done by Joris IJsselmuiden and Tim Samshuijzen, again guided by Rineke Verbrugge, Implementing the adaptive diary assistant in OutOfBrain is also a joint effort by Joris IJsselmuiden and Tim Samshuijzen. The programming environment OutOfBrain is a creation of Tim Samshuijzen. Joris IJsselmuiden is the author of this Master's thesis, supervised and guided by Rineke Verbrugge.

OutOfBrain manual: Appendix A
OutOfBrain website: www.outofbrain.com

Legal notice

The ideas and concepts presented in this document that are directly related to OutOfBrain are protected by international copyright laws. The unique use of run-time graph rewriting in combination with integrated agents has not yet been patented. This document is to be treated as an international publication, and hence the ideas and concepts appearing in it may not be patented by any other organisation at a later stage. OutOfBrain software is not shareware or freeware. OutOfBrain may not be used or distributed without either a licence or written consent obtained from Rockingstone IT B.V.

List of Figures

1.1 Simple OutOfBrain rewrite rule 15
2.1 Scheme for deliberation and means-end-reasoning 17
2.2 Example preference ordering (drive-rain-umbrella) 21
2.3 Example BDI model (coin-toss game) 30
2.4 Deliberation and means-end-reasoning in BOID 34
3.1 Graph rewriting ontology 40
3.2 The packet world 45
3.3 OutOfBrain ontology 47
4.1 The diary assistant's seven components 55
4.2 QDT example (overcast-rain-umbrella) 58
4.3 Preview of the diary-agent's functionality 63
4.4 Three agents, a blackboard and an interface 64
4.5 The classes Item and Message 67
4.6 Communication during planning 70
4.7 Communication during distance-matrix maintenance 72
4.8 Interface screenshot 74
5.1 Root of the diary-agent 77
5.2 Diary-agent branch 1: calendar maintenance 78
5.3 Pseudo-code equivalent to diary-agent branch 1 78
5.4 Diary-agent branch 2: suggestion generation 79
5.5 Diary-agent branch 3: type-tree update 82
5.6 Diary-agent branch 4: user-energy update 83
5.7 Snapshot of OutOfBrain code: user-agent's root 86
5.8 Todo-agent's preferences 87
5.9 User-agent's preferences 88
5.10 Source code of two preferences 89
B.1 Todo-agent 153
B.2 User-agent's root 153
B.3 User-agent branch 1: select best suggestion 154
B.4 User-agent branch 2: retrieve a distance 154
B.5 User-agent branch 3: retrieve attribute values 154
B.6 User-agent branch 4: update user-energy 155
B.7 User-agent branch 5: update type-tree 155
C.1 Example todo-item 156
C.2 Example do-item 157
C.3 Example message 157
C.4 Example distance-matrix 158
C.5 Example type-tree 159
C.6 Example user-energy 160

List of Tables

4.1 Normality ranks for the overcast-rain-umbrella example 58
4.2 Preference ranks for the overcast-rain-umbrella example 58
4.3 Expected utility ranks for the overcast-rain-umbrella example 59
4.4 Expected utility ranks for the augmented overcast-rain-umbrella example 60
5.1 The do-items from the test 90
5.2 The todo-items from the test 90

1 Introduction

People in the twenty-first century are busy. Sometimes it is hard to keep track of all one's appointments and obligations. Electronic diaries on personal digital assistants (PDA's) are quite popular nowadays and this trend will probably persist. Our aim is to develop an adaptive diary assistant that offers a bit more than existing electronic diaries. In this system, we incorporate a variety of different techniques: qualitative decision theory, graph rewriting, agent communication, user modelling and interface design. This adaptive diary assistant serves as a test-bed for the agent programming environment OutOfBrain combined with a qualitative decision architecture.

This Master's thesis consists of six Chapters. The first one gives general information about the project and provides the reader with some insight into our motivations, goals and applied techniques. Chapter 2 covers the studies that inspired our qualitative decision architecture. Chapter 3 treats some literature and our own insights on graph rewriting, agent communication, user modelling and interface design. Chapter 4 explains our own model and logic (derived from literature) as well as the design for the adaptive diary assistant. The fifth Chapter presents the implementation of this model and logic in the programming environment OutOfBrain and the application we build in that programming environment. Finally, Chapter 6 sheds some light on our conclusions, the evaluation of our goals and ideas for future research. The Appendices provide extra information on the programming environment OutOfBrain. Appendix A is the current version of the OutOfBrain manual, Appendix B is an overview of the diary assistant's implementation and finally, Appendix C sheds some light on the host graph the agents have to operate.

The present Chapter provides the reader with an overview of the motivations, goals and applied techniques of the project. Section 1.1 shows why the project is relevant to artificial intelligence and the field of multiagent systems (MAS). In Section 1.2, our goals are formulated and Section 1.3 gives an overview of the techniques we use.

1.1 Project motivations

The last fifteen years or so, multiagent systems (MAS) have received ever increasing attention from both science and business. In science, MAS already has an established role in the field of modelling and simulation. When mimicking biological agents it is a straightforward choice to use software agents or robots. A system consisting of more than one biological agent satisfies the demands that make the use of MAS sensible. In such a system, there is cooperation and/or competition, agents can have different goals, and knowledge is distributed among them. The scientific community is interested in the study of agents for two reasons. First, it helps us to better understand the behaviour and mental states of biological agents such as humans. Besides artificial intelligence, the studies of biology, psychology, philosophy and language are the main players in this quest. Second, it can aid the field of computer science in

the development of distributed systems and convenient programming metaphors. These goals do not contradict each other; in many studies both of them are pursued.

Analogously, one can distinguish between two approaches to the development of a MAS. Some researchers start with studying biological agents, formalize their properties and implement them in a simplified simulation. These researchers are concerned with biological plausibility. Others start with studying computational systems like programming languages and logic and try to create agents from there. These people are interested in formal properties like soundness, completeness and tractability. Again, a combination of these approaches is quite common, leading to biologically plausible agents that make use of some of the most powerful tools in computer science. In the process of creation it is often a good idea to take nature as an example. In many cases, evolution has much better solutions than we can come up with ourselves.

MAS has not had a wide influence on practical applications yet [Luck et al. 2005, Wooldridge 2002]. Only a very narrow part of the software spectrum actually uses multiagent systems[1]. MAS is simply not useful in most applications, because these applications do not have distributed knowledge, differing goals and cooperation or competition. We try to build a multiagent system that has at least some of these characteristics. In the meantime, we aim to improve the agent programming environment OutOfBrain.

The ALICE research group of the Artificial Intelligence department at the University of Groningen (R*u*G) has been studying multiagent systems for years. They are especially interested in the study of social structures, in particular teams of agents, societies of agents and competitive settings like Robocup. We hope that the ALICE research group can benefit from this project. In particular, we believe that our work will shed some light on the practical application of a multiagent team, since we create a team of agents that has the collective intention to help the user maintain his or her diary. We refer to [Dunin-Kęplicz and Verbrugge 2002] for a formal explanation of collective intentions.

There is an increasing desire for computer programs with realistic and clearly visible intelligence. Good examples of realistic, visible intelligence are the chatbots that have been created in the fourth quarter of the twentieth century (e.g. the award winning ALICE chatbot[2]). We try to reach such intelligence through a combination of a diverse set of techniques: qualitative decision theory, graph rewriting, agent communication, user modelling and interface design. None of these techniques is in itself new or particularly innovative. It is the combination of all of them that makes this project unique. We expect that by choosing such a wide variety of techniques, we incorporate an abundance of expressive power, leading to realistic and clearly visible intelligence.

This goal of "real" intelligence in a computer is also one of the reasons Rockingstone participates in this project. They are developing a programming environment particularly suitable for artificial intelligence programming: OutOfBrain. Rockingstone wants to make sure that OutOfBrain obeys the guide-lines set out by the academic MAS community, hence the cooperation with the University of Groningen. Another motivation of Rockingstone to participate in this project is that they want to guide the further development of OutOfBrain by incremental testing. Possible improvements to the environment emerge while designing and implementing the adaptive diary assistant.

[1] Multiagent systems are used more and more in software engineering as a useful programming metaphor. It often helps to think of objects as agents. However, this is not the usage of MAS we are aiming at here. We are mainly interested in true MAS, with distributed knowledge, differing goals and cooperation or competition.

[2] That this bot is called ALICE, just like the artificial intelligence research group at the University of Groningen, is a coincidence.

1.2 Project goals

We examine how qualitative decision theory, graph rewriting, agent communication, user modelling and interface design can be combined in a practical application. Through this project, we want to improve our own understanding of these topics, with emphasis on qualitative decision theory and graph rewriting. We also hope to contribute something to the corresponding research fields.

OutOfBrain is a response to an increasing demand by the software development community for a good agent programming language. We aim to improve OutOfBrain by incorporating theories on agent behaviour derived from literature. Thanks to the current project, the developers of OutOfBrain can get a clear picture of how the programming environment behaves during the development of a real application. To do this, we need an appropriate test-bed. An adaptive diary assistant is suitable for several reasons. First of all, the domain of a diary system is well defined and relatively small, making it an appropriate medium for theoretical research while being more than just a toy-problem. Furthermore, the use of mental attitudes makes sense in an adaptive diary assistant. An agent that models the user in order to maintain his or her diary could benefit greatly from natural concepts like beliefs and intentions. Another reason that makes the domain suitable is the fact that there are many opportunities for adaptation. The application we try to develop is a personal assistant that maintains models of the user as well as the world around him or her.

To make our goals more explicit, it is best to split it into two parts. The first more or less represents the motivation from Artificial Intelligence at the University of Groningen. We want to get a better understanding of the issues a developer has to face when creating a multiagent system. To achieve this, we try to create a team of agents that uses qualitative decision theory in a dynamical environment and incorporate it into an adaptive diary assistant. To highlight the focus of our research, we define our first research question as follows.

- **How can Boutilier's QDT architecture be extended to allow for temporal reasoning?**

Our second goal corresponds roughly to the motivations of Rockingstone. We want to explore and extend the capabilities of the OutOfBrain system. To do this, we try to create a team of agents that shows realistic and clearly visible intelligence. Our multiagent architecture, supported by literature, is translated to an OutOfBrain model that is in turn used to implement an adaptive diary assistant. What follows is the second research question we focus on.

- **Can Boutilier's QDT architecture, extended with a temporal modality, be integrated into OutOfBrain?**

These two main goals proved to be quite consistent, which made for a smooth cooperation. The university provides the project with the necessary theoretical framework. Rockingstone makes implementation of the theory in a versatile environment possible. The residue of our goals when we remove these two parts is concerned with agent communication, user modelling and interface design. These matters are treated as secondary goals. They should be viewed as the context in which qualitative decision theory and graph rewriting operate. Communication between agents, modelling the user and a good interface should be able to increase the usability of the diary assistant in several ways.

First of all, through the study of agent communication, we want to increase the application's performance and get a better understanding of sociality in MAS. Social skills are necessary in most MAS applications (see for example [Weyns and Holvoet 2004] or [Stone

and Veloso 2000]). Second, we study user modelling, because literature shows that adaptation skills improve performance on most complex tasks [Stone and Veloso 2000]. We think that the ability to learn should not be considered as special. It should form an integral part of any agent in a complex world. Finally, our reason for studying interface design is simple. We think that every software project should devote at least some attention to the design of a proper interface.

Creating a useful piece of software is also treated as a secondary goal. It is not so much where the journey ends but it is the journey itself we are interested in. However, throughout the duration of this project, we keep the development of a commercially interesting diary assistant in mind. Our application lacks two important features which make it less interesting as a product. First, a diary assistant should be portable. If the PC version turns out to be a success, it would not be a major undertaking to create a PDA version from it though. Second, the diary assistant should be connected to other PDA's. Most companies and other institutions have a central diary system so that appointments can be scheduled more efficiently. When combined with portable devices, connectedness becomes a powerful tool. MAS will play an important role in the emerging field of ambient intelligence [Aarts and Marzano 2003]. Soon, portable, interconnected personal assistants will be a common sight, each one represented by an autonomous agent. Besides maintaining one's diary, the assistant can perform all sorts of other tasks.

1.3 Applied techniques

We aim to build an adaptive diary assistant using a unique combination of techniques from different fields of research. The techniques we use can be divided into three groups: qualitative decision theory, graph rewriting and the remainder: agent communication, user modelling and interface design. All these techniques will be treated later in the thesis and they are introduced briefly below in Sections 1.3.1 through 1.3.5. Our main focus is on qualitative decision theory and graph rewriting.

The research protocol (methods) of this project can best be classified as scientific software development. The protocol is an iterative cycle running from literature study to design and finally to implementation. During the course of the project, the focus shifts ever further toward the back end of the cycle (implementation). In parallel to this, the writing of the thesis takes place under extensive guidance and supervision. Furthermore, we perform weekly brainstorm sessions to plan our activities, gain new insights and stay on the same track.

Because we venture in relatively uncharted territory, a lot of work has to be done before we can actually start designing. During the first stage, by studying the literature, we try to pin down the current state of affairs in the scientific community, concerning the topics that are of interest to the project. Then, we try to extract ideas from literature that could be useful during design and implementation. During the second stage, we aim to design three things. First, we create a MAS architecture composed of QDT and parts of BDI. Second, we specialize this design into one that is suitable for implementation in OutOfBrain. Finally, we produce a design for the adaptive diary assistant we plan to build. Implementation can be divided into three tasks: First, we build the MAS architecture in Delphi so that it becomes an integral part of the OutOfBrain environment. Second, we implement the adaptive diary assistant in OutOfBrain. Consequently, two totally different forms of programming have to take place: one is textual, the other is graphical and at a higher level of abstraction.

Before briefly introducing the applied techniques, we should shortly explain the application we have in mind. The diary assistant is composed of a diary and a todo-list. The

diary is used to store appointments and other obligations that have a specific date and time associated with them. The todo-list is used to store obligations that are not associated with a specific date or time. The system should move these todo-items to a specific point in time on the diary, bearing in mind the workload and other appointments of the user. Also, instead of manually scheduling things directly into the diary, obligations can be put on the todo-list along with the command to schedule it automatically.

The longer the user works with the diary assistant, the better the assistant will anticipate on his or her needs. For example, the diary assistant could ask itself the following questions.

- Which hours of the week would this user like to be free?
- How much energy does the user have on a typical Wednesday?
- How many meetings can the user handle in one week?
- Which type of activity is relatively exhausting for the user?
- How many items should there be on the todo-list before I start moving them to the diary?
- Which activities cannot be done while the user is out of the country?
- How long is the trip from home to the office?

Now that there is some insight in the intended functionality of the application, the applied techniques should be introduced one by one.

1.3.1 Qualitative decision theory

Our multiagent architecture is based on QDT (Section 2.3) [Boutilier 1994]. This architecture uses a bimodal logic and ranks possible worlds in preference and normality orderings. We also incorporate components of BDI (Section 2.4) [Rao and Georgeff 1991], in particular the notion of time and the handling of actions and other events. We use qualitative representations whenever we can, because of the drawbacks of quantitative methods (see Section 2.2). Nonetheless, some information is best represented quantitatively, for example the dates and times in a diary. In our architecture, composed of QDT and BDI, an agent can reason about preferences, normalities, time, actions and other events. The adaptive diary assistant we build in this architecture has three agents: One modelling the user, one maintaining the user's diary and one maintaining the todo-list. This team of agents has the collective intention to serve the user [Dunin-Kęplicz and Verbrugge 2002].

1.3.2 Graph rewriting and OutOfBrain

The programming environment OutOfBrain is currently under development at Rockingstone. Its manual can be found in Appendix A. The environment is designed specifically for the programming of intelligent agents. Another fact that distinguishes it from other programming languages is that it uses graphical representations instead of text. OutOfBrain is a high level programming language built on Delphi, the same language we use for the implementation of the diary assistant's interface. The mechanisms at work in the OutOfBrain system have a lot in common with graph rewriting. Therefore we compare it to the literature on existing graph rewriting techniques in Section 3.1 [Blostein et al. 1995].

Figure 1.1 shows a simple example of a rewrite rule in OutOfBrain. The search algorithm looks for a subgraph in the hostgraph that is isomorphic to the premise of the implication and replaces it with the implication's conclusion. OutOfBrain has much in

common with logic programming languages like PROLOG. The main differences between OutOfBrain and PROLOG are that OutOfBrain uses graphical representations and a different, but equally powerful search engine.

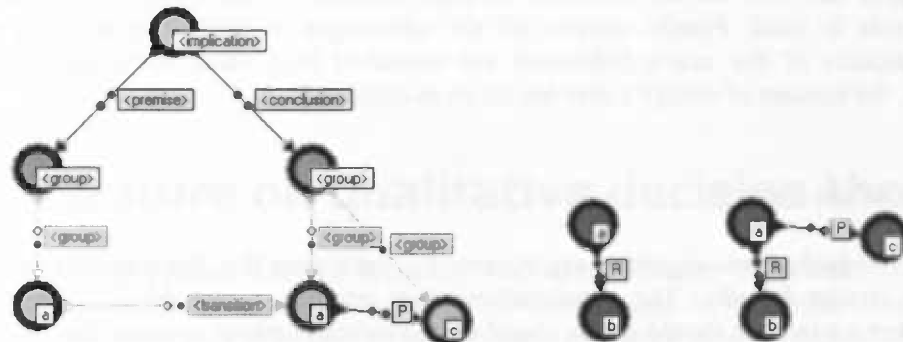


Figure 1.1: A simple rewrite rule in OutOfBrain (left) and an appropriate hostgraph (middle) and resultgraph (right). On screen, different colours are used to improve usability.

1.3.3 Agent communication

Communication is a crucial part of most multiagent systems. We use the FIPA standard for agent communication [Wooldridge 2002, FIPA 2001]. In this standard, a message has the following attributes: The performative (type of the message, e.g. request, inform), the sender, the receiver, the content and the language and ontology that should be used to interpret the message. This standard helps the agents to assist each other in pursuing the collective intention of serving the user [Dignum et al. 2001]. It also aids them in developing a common frame of reference.

1.3.4 User modelling

There are many methods at our disposal to facilitate user modelling. After careful consideration, we ended up with four different tools that work together to model the user: reordering and information injection in possible world orderings, qualitative reinforcement learning, production compilation and statistical analysis.

QDT, the qualitative decision architecture our system is based on, is designed in such a way that it is compatible with learning and other dynamical processes. Learning amounts to the reordering of preference and normality orderings on possible worlds as well as the injection of new information into them. Two sources of information could be used as input for this learning mechanism.

First, we incorporate a qualitative version of reinforcement learning. The diary assistant can be rewarded or punished for its behaviour, either implicitly or explicitly. For example, if the user is not satisfied by the diary assistant's suggestions, he or she can say so. The system will then try to find out why the user is not satisfied using multiple choice questions[1]. Second, we could use a mechanism based on production compilation in ACT-R [Taatgen and Lee 2003]. This architecture can be used to deduce rules from history. Just like

[1] This learning behaviour can be shut down so that the user is not burdened with questions. He or she could be in a hurry or simply not in the mood for questions. We use multiple choice questions because natural language processing is beyond the scope of this project.

QDT, it has a strong cognitive basis since it was developed for cognitive modelling. If an agent discovers that event A has been succeeded by event B several times, the (default) rule $A \rightarrow B$ should be added to the agent's beliefs. Rules can be removed or augmented in the same way. Again, the user can be consulted through multiple choice questions to make sure a derived rule is valid. Finally, despite all the advantages of qualitative learning methods, certain aspects of the user's behaviour are modelled best using statistical analysis, for example, the amount of energy a user has on an average day.

1.3.5 Interface design

We use standards from cognitive ergonomics to make sure that the interface of the diary assistant is user friendly. The conversational text-interface is an old-fashioned means of control, but we think it should not be abandoned. Modern chatbots show us that if an agent is sufficiently intelligent, a conversational text-interface is a convenient medium of control. However, we choose to support the conversational text-interface with graphical control mechanisms like buttons. Natural language processing (NLP) makes a conversational text-interface truly useful. If NLP were integrated into the system, the use of buttons and the like would not be necessary to ensure a user friendly interface. Unfortunately, this is beyond the scope of our project. The development of a PDA version of the diary assistant would require extensive study in interface design and cognitive ergonomics, since PDA's have small screens and the user controls them differently. In the current project however, we are not yet concerned with these issues.

Now that our motivations, goals and applied techniques have been explained, we move on the literature we studied for this project.

2 Literature on qualitative decision theory

The literature is divided into two Chapters. The current Chapter treats studies on qualitative decision theory. It is the topic receiving our main focus. In particular Boutilier's QDT (Section 2.3) takes up a large part of this Chapter. Literature on other topics than qualitative decision theory is treated in Chapter 3. These are graph rewriting, agent communication, user modelling and interface design.

In Multiagent Systems, MAS from now on, agents reason about what state of affairs they want to achieve and how they should go about achieving this state. Typically an iteration of an agent's reasoning can be divided into two main parts: *deliberation* (to determine the goal) and *means-end-reasoning* (how to achieve this goal) [Dunin-Kęplicz and Verbrugge 2002]. Figure 2.1, taken from [Broersen et al. 2002], shows the typical steps an agent has to take to reach a certain goal. First, the agent observes the environment. He uses the observations to generate a candidate goal set. Next, he selects one of these goals through a reasoning process. The goal that comes out on top is used in the planning module to determine which actions the agent should execute. The border between deliberation and means-end-reasoning lies where the chosen goal enters the planning module. This thesis does not deal with means-end-reasoning. At least not in its literature study. We refer to [Boella et al. 2005, Boutilier 1994, Dunin-Kęplicz and Verbrugge 2004] for means-end-reasoning in the context of qualitative decision theory.

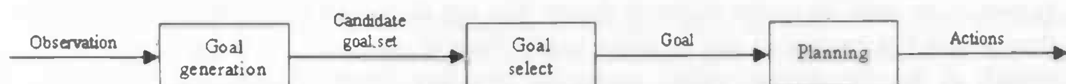


Figure 2.1: Typical scheme for deliberation and means-end-reasoning in an agent. The border between them lies where the chosen goal enters the planning module.

According to [Dastani et al. 2003], there are four successful branches in decision theory: *classical decision theory (CDT)*, *qualitative decision theory (QDT)*[1], *knowledge based systems (KBS)* and *beliefs, desires and intentions systems (BDI)*. CDT is the most established framework of these four. It is much older than the others and it is distinguished from QDT, KBS and BDI by the fact that it uses real valued calculations for determining which states to pursue (Section 2.1). The other three all use less of a quantitative method, hence they are called qualitative decision theories. Our study focuses on QDT and BDI systems (two of the qualitative decision theories).

[1] Dastani et al. use the abbreviation QDT to refer to Boutilier's qualitative decision theory, not to refer to qualitative decision methods in general. We adopt this convention. We use QDT to refer to Boutilier's work and qualitative decision theory to refer to the field in general.

In Section 2.1 we describe the global history of decision theory derived from [Doyle and Thomason 1999]. Section 2.2 summarizes the reasons for abandoning CDT when modelling complex problem solvers. Most of these reasons have again been derived from [Doyle and Thomason 1999]. Section 2.3 covers QDT in detail [Boutilier 1994]. This work is the main inspiration for our own architecture described in Chapter 4. Section 2.4 summarizes [Rao and Georgeff 1991] on the BDI architecture. We use this study to complement our model with some concepts that are not present in QDT. Also, in Section 2.4.3, we present an overview of the BOID architecture [Broersen et al. 2002], a direct cousin of BDI. Finally, Section 2.5 presents some thoughts on how to combine QDT and BDI into a single model.

2.1 Some history on decision theory

Decision making has been studied formally since the seventeenth century. The traditional, quantitative approach to automatic decision making reached maturity in the 1950s [Doyle and Thomason 1999]. A cornerstone in the development of quantitative decision theory was [Ramsey 1926]. Frank Ramsey's theory enabled us to derive sensible, quantitative utilities and probabilities over alternatives. Ramsey's most notable successor is arguably Leonard Savage [Savage 1954]. *Modern Bayesian decision theory* or CDT was born. It provided a way to choose between alternatives, maximizing *expected utility*. The expected utility of an action α is calculated as follows.

$$E(\alpha) = \sum_{i=1}^{n} U(\omega_i)P(\omega_i \mid \alpha)$$

Where n is the number of possible outcomes of α, $U(\omega_i)$ is the utility function over outcome ω_i and $P(\omega_i|\alpha)$ is the probability that action α leads to outcome ω_i.

Almost immediately after the birth of modern Bayesian decision theory, Herbert Simon and other artificial intelligence researchers questioned the usefulness of the new theory. They argued that it requires too much information, reasoning power and memory. However, modern Bayesian decision theory was not discarded due to this critique. In fact, the theory still has numerous applications today. Instead, the critique led to the birth of a new branch of decision theory called qualitative decision theory [Doyle and Thomason 1999], using preference orderings and likelihood orderings on possible worlds or even binary values to represent preference and likelihood. Many of the problems of modern Bayesian decision theory do not apply to this new branch (see Section 2.2).

It is time for a simple example to explain informally how these qualitative methods work. Imagine an agent who models his user, living in Helsinki, to help him or her with planning appointments. He could deduce that he should not go visit his mother in northern Finland. He desires to go, but he also believes that he would not be back in time for an important meeting next Monday, because the bush pilot flies back from northern Finland to Helsinki only once a week. The agent desires to close an important business deal during the meeting on Monday and he believes he can probably achieve this too. If this deal is too important for him, the agent will decide not to go to his mother. The central concepts in this deduction are the verbs desire and believe. The argument could quite easily be translated into a logic for qualitative decision making. This reflects the intuitive nature of such logics.

In 1987, Michael Bratman postulated that intentions are an important concept in human practical reasoning and that they are not reducible to beliefs and desires [Bratman 1987]. His views are now widely accepted and used extensively in qualitative decision theory,

most notably in BDI, the most popular approach among qualitative decision theories. An agent's intentions are a consistent subset of his goals; the goals that currently have the agent's focus [Dunin-Kęplicz and Verbrugge 2002].

In [Cohen and Levesque 1990], a logic is explained that uses statements about time series as well as about an agent's goals and beliefs. Intentions are not considered to be fundamental in their logic. Instead they are defined as choice with commitment. Cohen and Levesque's work has made a large impact on the field of decision theory. However, it has been criticized to be too complicated (in terms of complexity measures and comprehensiveness). A comparable study has been conducted by Rao and Georgeff, which is explained in detail in Section 2.4. Their BDI architecture does treat intentions to be fundamental. The theory they ended up with is considered to be easier to apply than the theory by Cohen and Levesque for several reasons. We will not get into these reasons in this Master's thesis however.

2.2 Motivations for using qualitative methods

There are many reasons to prefer qualitative methods over quantitative ones. [Doyle and Thomason 1999] provides a nice overview. When people give each other advice, they do so by means of symbolic communication. Human reasoning also depends largely on symbolic representations[1], most human reasoning procedures do not use numeric trade-offs. Qualitative decision theory provides a better way of describing the reasons for decisions in ways that humans find intelligible than modern Bayesian decision theory. According to [Fong et al. 2003] and many other studies, interaction performance increases when robots behave more like humans. Qualitative decision methods can contribute to the resemblance between human behaviour and robot or software-agent behaviour. The diary assistant we want to develop should give the user advice about task planning much like a human secretary would. This makes qualitative representations a natural choice for facilitating both communication and reasoning. Traditional quantitative methods for making decisions are not supported by either intuition or biological plausibility. This becomes even more apparent when one considers the question whether there is a universal currency for measuring the utility of outcomes [Doyle and Thomason 1999].

Besides these philosophical objections, there are also some more practical issues involved[2]. First of all, it is hard to determine all these utilities and probabilities. Together with time and space complexity, this makes accurate deliberation intractable in most complex domains. Already in the 1950s, Herbert Simon and his colleagues proved that for many complex problems, quantitative decision theory requires too much information, reasoning power and memory [Doyle and Thomason 1999]. Due to the unsatisfactory results of CDT in complex environments, a new approach to problem solving emerged: *bounded rationality* or *satisficing* [Simon 1987]. Instead of pursuing a state that is guaranteed to be the best one possible, this new approach pursues a possibly suboptimal state. An agent will only reason for a certain period of time and then tries to achieve the state it has deliberated to be "pretty good". The agent simply doesn't have the time to think it over more thoroughly. Generating all possible plans and then selecting the best one would lead to optimal solutions. However the number of possible plans grows exponentially with the number of possible actions, making this method infeasible. Therefore in complex domains it is better to first derive goals

[1] Although these symbolic representations most probably have a distributed realisation basis in our brains.
[2] Note that the philosophical objections described here can have very practical consequences.

and then construct plans that lead to those worlds despite the risk of suboptimal results [Broersen et al. 2002].

There are other problems concerning the broad applicability of quantitative methods. First of all, they lack creativity in the sense that they cannot make decisions in unforeseen settings. Furthermore, most quantitative methods do not provide appropriate ways to make decisions when there is partial information or uncertainty about goals and preferences, whereas most qualitative methods do. Quantitative methods also lack ways of handling new information or the ability to reconsider goals.

The reconsideration of goals is the source of an important debate in the history of artificial intelligence. When should an agent check whether his goals are still worth pursuing? Possible reasons for abandoning goals are that the goal or its motivation has been satisfied, that the goal or its motivation can no longer be achieved or that its motivation may be reached by some other means. Goals must be reinforced during reconsideration or they are dropped [Broersen et al. 2002]. The problem is that goal reconsideration takes up valuable resources, so it should only be done when the agent expects it is necessary.

As mentioned before, quantitative methods do not provide appropriate ways for capturing human expressions about decision making. One of the most important types of expressions used in human decision making is the *generic preference* relation, for example: "I would rather do the dishes tomorrow than today." or "I prefer to have no obligations between 1 pm and 2 pm." The last reason we put forward here for preferring qualitative methods over quantitative ones is the requirement for broad knowledge of the world. It is much easier to construct a database of world knowledge without the tedious consideration of all the utilities and probabilities that hold in the domain.

Despite all the drawbacks of quantitative methods, many types of information are simply best represented quantitatively. There have been many studies that combine quantitative and qualitative information into hybrid systems. Furthermore, the developers of qualitative methods tend to build on existing quantitative techniques whenever possible.

2.3 QDT logic

The theory [Mehdi Dastani 2003] refers to as QDT is presented in [Boutilier 1994]. In this paper, Craig Boutilier describes a logic CO and a corresponding possible worlds model, along with several reasoning strategies. In Boutilier's theory, a goal is generally a proposition or formula that the agent should make true. The corresponding semantics consist of a set of possible worlds or states that satisfy the goal proposition [Cohen and Levesque 1990]. Note that there is only one agent in this framework. As we shall see, Boutilier's theory is easily expandable to a version that includes more than one agent.

2.3.1 Introduction to QDT

Sometimes, goals turn out to be unachievable. An agent should be able to adopt alternative goals in such cases. Furthermore, an agent should be allowed to formulate exceptional situations in which a certain goal is less desirable. In order to achieve all this, QDT ranks possible worlds according to their degree of preference. The basic concept in the theory is $I(B|A)$, which means "ideally B, given A". The language L_{CPL} Boutilier uses consists of a finite set of propositional variables P, the usual connectives and the unary modal operators $\Box$

and $\bar{\Box}$. Boutilier uses A and B for propositional atoms and α and β for formulas[1]. The operator □ should be read as 'preferably', whereas $\bar{\Box}$ should be interpreted as 'less preferred'. The theory is not concerned with the actions that actually take an agent to a goal-world. The possible worlds semantics for $\mathbf{L}_{CPL}$ consists of models of the form $M = \langle W, \leq, V \rangle$, where W is the set of possible worlds, $\leq$ is a transitive, connected binary relation on W^2 and V is the valuation function mapping propositions in possible worlds to binary truth values ($V: P \times W \rightarrow \{0, 1\}$). The relation $v \leq w$ means that world v is at least as preferred as world w.

Boutilier illustrates preference orderings with the example in Figure 2.2. According to this ordering, the agent prefers worlds in which he is not driving to work ($\neg D$), it is not raining ($\neg R$) and he is not carrying an umbrella ($\neg U$), the bottom world in Figure 2.2. If this is not possible, he prefers worlds where he is driving to work, it is raining, but he is not carrying an umbrella. This world could occur, because the agent cannot control whether it is raining or not. Even less preferred are worlds in which the agent is walking in the rain with an umbrella, this situation might occur because the agent's car is broken. The least preferred world in this example is the one in which the agent is walking in the rain without an umbrella. Note that preference orderings need not contain complete knowledge, hence the term "cluster of possible worlds". If nothing is stated about the truth value of a proposition, the agent is indifferent to the truth of that proposition. This property has been derived from *system Z* [Pearl 1990]. The ability to reason with incomplete knowledge is a necessity in complex domains.

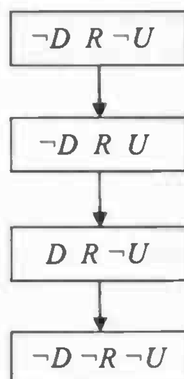


Figure 2.2: Example preference ordering from [Boutilier 1994]. The possible worlds cluster at the bottom represents the best worlds. The agent prefers worlds in which he is not driving to work, it is not raining and he is not carrying an umbrella. If this is not possible, he prefers a world where he is driving, it is raining, but he is not carrying an umbrella.

Boutilier's preference ordering scheme is a lot like constraint satisfaction. The agent tries to satisfy as many constraints as possible, ending up with the best possible world. In many cases, the ideal situation is not achievable, because the agent does not have the power to manipulate the truth values of certain propositions (e.g. whether it is raining or not). We now turn to the truth definitions for Boutilier's modal operators:

Def 2.1 $\quad M \models_w \Box\alpha$ iff for each v such that $v \leq w$, $M \models_v \alpha$

[1] Although Boutilier does not say this explicitly; it follows more or less, from the formulas in his 1994 paper. He does not provide a clear definition of the language $\mathbf{L}_{CPL}$.

[2] W consists of clusters of equally preferred worlds and these clusters are totally and transitively ordered by $\leq$. When the logic CO is extended to the logic QDT, $\leq$ is replaced by $\leq_P$, because another binary relation $\leq_N$ is needed in QDT.

Def 2.2 $M \models_w \overleftarrow{\Box}\alpha$ iff for each v such that $w < v$, $M \models_v \alpha$

Definition 2.1 states that $\Box\alpha$ is true at world w if and only if α is true in every world at least as preferred as w. Definition 2.2 tells us that $\overleftarrow{\Box}\alpha$ is true in w just in case α is true in every world less preferred than w. These modal operators are used to reason about worlds on a preference ordering, in the same way that they can be used to reason about worlds on a time sequence, either to look forward or backward in time. Several modal operators can be derived from these two basic ones in the usual way:

Def 2.3 $M \models_w \Diamond\alpha$ iff $M \models_w \neg\Box\neg\alpha$

Def 2.4 $M \models_w \overleftarrow{\Diamond}\alpha$ iff $M \models_w \neg\overleftarrow{\Box}\neg\alpha$

Def 2.5 $M \models_w \overleftrightarrow{\Box}\alpha$ iff $M \models_w \Box\alpha$ and $M \models_w \overleftarrow{\Box}\alpha$

Def 2.6 $M \models_w \overleftrightarrow{\Diamond}\alpha$ iff $M \models_w \Diamond\alpha$ or $M \models_w \overleftarrow{\Diamond}\alpha$

It is common practice to define the diamond operator ($\Diamond$) as not-box-not ($\neg\Box\neg$). According to Definition 2.3, $M \models_w \Diamond\alpha$ means that α is true in at least one world at least as preferred as w. Definition 2.4 expresses that $\overleftarrow{\Diamond}\alpha$ corresponds to the fact that α is true in at least one world that is less preferred. Definition 2.5 says that $\overleftrightarrow{\Box}\alpha$ corresponds to the fact that α is true in all worlds in the preference ordering, whereas Definition 2.6 says that $\overleftrightarrow{\Diamond}\alpha$ is the same as stating that α is true in at least one world in the preference ordering.

2.3.2 The logic CO

An approximation of the meaning of $I(B|A)$ is "In the most preferred A-worlds, B should also hold" (Definition 2.7). This is defined semantically as follows. A is false in each and every cluster of possible worlds on the preference ordering or A is true in at least one cluster of possible worlds on the preference ordering. In the latter case, B has to be true in each equally or more preferred A-world. This comes down to the fact that the agent pursues the truth of B when A is true. This definition resembles the one stated by David Lewis about counter-factuals [Lewis 1973]. Reasoning with counter-factuals is concerned with what the world would look like if a given sentence were false. For example, what would the world be like if John F. Kennedy had not been shot? In Lewis' semantics, the sentence "If Kennedy had not been shot ($\neg K$), the USA would be a peaceful country (P)" is true if and only if in the $\neg K$-worlds that most resemble the real world, P holds. If we substitute "the $\neg K$-worlds that most resemble the real world" with "the most preferred $\neg K$-worlds", we end up with Boutilier's definition for the ideally operator; $I(P|\neg K)$.

Def 2.7 $I(B|A) \equiv_{df} \overleftrightarrow{\Box}\neg A \vee \overleftrightarrow{\Diamond}(A \wedge \Box(A \rightarrow B))$

Craig Boutilier uses some extra notation that has to be mentioned here. Absolute preferences are represented as $I(A|\top)$ or $I(A)$ for short, which means that A is preferred in every context. $I(A)$ is in turn logically equivalent to $\overleftrightarrow{\Diamond}\Box A$. Intuitively, this means that from at least one cluster of worlds on the preference ordering, A is true in that cluster and every cluster that is more preferred (including the most preferred cluster). This is the same as saying that one has an absolute preference toward A.

One can also specify "don't care" conditions like $\neg I(\neg B|A)$: In the most preferred A-worlds it is not required that $\neg B$, or equivalently: "B is *tolerable* given A". Recall that the diamond operator has the same meaning as not-box-not, the same is true for tolerable and not-required-not. The formula $\neg I(\neg B|A)$ is abbreviated as $T(B|A)$.

Instead of comparing the preference relation between worlds, one can compare two propositions directly. $A \leq B$ means that the best A-worlds are at least as good as the best B-worlds, see Definition 2.8. An agent has a preference of A over B if and only if everywhere on the preference ordering the following holds: If B is true in a world then A is true in a world that is at least as preferred as that world. Statements like this, as well as statements built from Definition 2.7, should be used to inject information about preferable propositions into a knowledge base in order to reason about preference orderings.

Def 2.8 $\quad A \leq B \equiv_{df} \overleftrightarrow{\Box}(B \rightarrow \Diamond A)$

In many domains it is useful to state strict preferences. We speak of a strict preference if a proposition is more desirable than its negation in every context. This situation is formalized in Definition 2.9. Intuitively, it means that everywhere on the preference ordering, if C is true in a world, C is also true in all more preferred worlds. Note that the closed world assumption is at work here, since $\neg C$ does not even occur in Definition 2.9. We simply assume that C is false if the truth of C is not explicitly stated.

Of course, preference information should be consistent. If a proposition is preferred in a certain situation, the proposition should also be tolerable in that situation (Definition 2.10). Furthermore, the property of *preferential detachment* holds in CO (Definition 2.11). If ideally B given A and ideally A in every context, then also ideally B in every context. This is an intuitive axiom: If A is true all over a certain bottom part of the preference ordering (most preferred worlds) and B should be true in the most preferred A-worlds, then B should be true in some bottom part of the ordering[1]. However, *factual detachment* (Definition 2.12) does not hold in CO, because there can always be a more preferred world than the one from which we evaluate Definition 2.12 where A is false (or unknown and thus false under the closed world assumption). Therefore B need not be true all over a certain bottom part of the preference ordering.

Def 2.9 $\quad \overleftrightarrow{\Box}(C \rightarrow \Box C)$ (true iff there is a strict preference for C)

Def 2.10 $\quad I(B|A) \rightarrow \neg I(\neg B|A)$ (axiom of CO)

Def 2.11 $\quad I(B|A) \wedge I(A) \rightarrow I(B)$ (axiom of CO)

Def 2.12 $\quad I(B|A) \wedge A \rightarrow I(B)$ (not an axiom of CO)

The most important feature of Boutilier's preference orderings is that an agent's preferences depend on the situation at hand. For example, an agent can have the following preferences at the same time: $I(U|R)$ and $I(\neg U|\neg R)$, representing that the agent should take an umbrella when it is raining and leave it home when it is not.

We will see in Section 2.3.3 that Boutilier uses a similar operator for normality orderings of possible worlds. These normality orderings tell the agents something about how likely a certain world is. This information should be combined with a preference ordering to

[1] A is true in some bottom part of the ordering if and only if $\overleftrightarrow{\Diamond}\Box A$ is true; there is an absolute preference for A. Note that A and B do not need to be true in the exact same bottom part.

determine the relative utility of a world, which in turn determines which world an agent should try to achieve. The preference and normality orderings also make default reasoning possible [Reiter 1980]; Boutilier's logic is defeasible. When new information becomes available, an agent can change his conclusions. Most logics (including BDI) are monotonic in the sense that once a conclusion has been reached, it cannot be retracted. Defeasibility is one of the reasons why we use QDT as a starting point for our model. Note that information injections about preference and normality relations can be done by the designer or by a learning process. For our adaptive diary assistant, we use Boutilier's preference ordering and normality ordering along with a third ordering that expresses time series. The apparatus for this third modality is derived from [Rao and Georgeff 1991].

2.3.3 The logic QDT

Boutilier extends his logic CO to the logic QDT. The CO models are supplemented by a relation $\leq_N$ on pairs of possible worlds and the symbol $\leq$ in CO is replaced by $\leq_P$ in QDT. The transitive, connected $\leq_N$-relation expresses normality orderings in the same way that the transitive, connected $\leq_P$-relation expresses preference orderings. $v \leq_N w$ means that world v is at least as likely to occur as world w. This extension makes it possible for an agent to consider only worlds that are likely to occur.

Combining the preference and normality ordering could lead to a method for considering only worlds that are both relatively likely to occur and relatively preferable. It seems that a mechanism similar to the maximization of expected utility [Savage 1954] could be used to determine goal worlds in Boutilier's system. One could link a natural number to the relative preference and relative normality of a world and multiply them into a measure of expected utility for that world. In his 1994 paper, Boutilier is very brief about how one should accomplish a measure of expected utility, but it seems to us that the method is straightforward and expected utility can be calculated much like Savage's method described in Section 2.1. The main difference then between CDT and QDT is that CDT uses real valued calculations where QDT uses natural numbers assigned to orderings. One could therefore question whether QDT is a real qualitative method. With respect to this question we propose to position QDT between CDT and BDI, because BDI uses purely binary representations and is therefore truly qualitative.

We now turn from Boutilier's logic CO to its extended version QDT. QDT-models are of the form M = $<W, \leq_P, \leq_N, V>$. W is the set of possible worlds, V is the corresponding valuation function assigning truth values to propositions in possible worlds. $\leq_P$ is a preference ordering on $W \times W$. Similarly, $\leq_N$ is a normality ordering on $W \times W$. The relation $v \leq_P w$ means that world v is at least as preferred as world w. The relation $v \leq_N w$ says that v is at least as normal a situation as w. A QDT-model is actually two different CO-models combined, one for preference, one for normality. Consequently, the logic QDT has twice as many modal operators as the logic CO: $\Box_P$, $\overleftarrow{\Box}_P$, $\Box_N$ and $\overleftarrow{\Box}_N$. These operators and their derived operators are interpreted as before (Definitions 2.1 through 2.6). The conditional $I(B|A)$ can be expressed using $\Box_P$ and $\overleftarrow{\Box}_P$ as in Definition 2.7. A new conditional connective $A \Rightarrow B$ in terms of $\Box_N$ and $\overleftarrow{\Box}_N$ is introduced to express the default rule "Under normal circumstances, if A, then also B."

The fact that QDT uses a non-monotonic logic, gives it an advantage over BDI with its monotonic logic. For an introduction to non-monotonic reasoning and default-logic, we refer to the classic [Reiter 1980]. The $\Rightarrow$-connective for default applications should also be interpreted using Definition 2.7. The definition for $A \Rightarrow B$ and $I(B|A)$ is the same, they only differ in their semantics. $I(B|A)$ means "ideally B given A" (see the more detailed description just above Definition 2.7). $A \Rightarrow B$ has the following meaning: "if A then normally also B".

This difference in meaning is due to the fact that they are applied to different orderings. When we examine Definition 2.7 applied to normality orderings instead of preference orderings, we get a description like this: *A* is false in each and every cluster of possible worlds on the normality ordering or *A* is true in at least one cluster of possible worlds on the normality ordering. In the latter case, *B* has to be true in each equally or more likely *A*-world. It is important to notice that Definition 2.8 through 2.12 and the axioms below also apply to normality orderings. The logic QDT truly is just two CO logics with exactly the same syntactical properties, but different semantics.

Boutilier uses the following axioms and inference rules for the logic QDT. They hold for both preference orderings and normality orderings, so the boxes and diamonds can have either a *P* as subscript or an *N* as subscript.

K $\Box(A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$

K' $\overleftarrow{\Box}(A \rightarrow B) \rightarrow (\overleftarrow{\Box} A \rightarrow \overleftarrow{\Box} B)$

T $\Box A \rightarrow A$

4 $\Box A \rightarrow \Box\Box A$

S $A \rightarrow \overleftarrow{\Box}\Diamond A$

H $\overleftrightarrow{\Diamond}(\Box A \wedge \overleftarrow{\Box} B) \rightarrow \overleftrightarrow{\Box}(A \vee B)$

Nec From A infer $\overleftrightarrow{\Box} A$

MP From $A \rightarrow B$ and A infer B

From the axioms and inference rules above, we only discuss the axioms **S** and **H**, because all other axioms are quite common in modal logic. Axiom **S** expresses the fact that if *A* is true in a world cluster *x* on the preference or normality ordering, then $\Diamond A$ is true in every cluster of worlds *y* less preferred or less likely than *x*. This means that *A* should be true in a cluster of worlds *z* at least as likely or preferable as *y*. This is an axiom, because *z* can be the same cluster as *x* where *A* is true[1]. Axiom **H** looks a bit more complicated, but it is a very intuitive axiom. If somewhere on the ordering, say in cluster *x*, $\Box A \wedge \overleftarrow{\Box} B$ is true, then $A \vee B$ is true everywhere on the ordering. After all, *A* is true in every cluster at least as preferred/likely as *x* and *B* is true in every cluster less preferred/likely than *x*.

2.3.4 Properties of QDT

A fair part of [Boutilier 1994] is about the properties of the logic QDT. First of all, he states that the logic is sound and complete with respect to the class of QDT-models. Next, he defines the default closure of a knowledge base as in Definition 2.13. *Cl(KB)* is the set of propositions that follow logically from applying default rules to the knowledge base. The propositions in *Cl(KB)* are all true under normal circumstances, but they can become false

---

[1] Axiom **S** is similar to the basic temporal axiom $A \rightarrow HFA$ [naam?]. It says that if *A* is true at a certain point in time, HFA is also true in that point. It has always been the case that at some time in the future *A* is true.

when new knowledge is added to *KB*. A QDT-agent ought to act as if *Cl(KB)* were true, not just *KB*. *Ideal goals* are defined by Definition 2.14.

Def 2.13 $Cl(KB) = \{\alpha \in \mathrm{L}_{CPL} : KB \Rightarrow \alpha\}$

Def 2.14 α is an ideal goal iff $M \models I(\alpha|Cl(KB))$

Events are divided into *controllables*, *influenceables* and *uncontrollables*. An example of an uncontrollable is whether it is raining or not. Taking an umbrella or leaving it at home is a typical controllable event. The set of controllable propositions is a subset of the influenceables. Consider a situation where α follows from either A, B or C: $M \models A \vee B \vee C \rightarrow \alpha$. If for example only B is controllable and A and C are uncontrollable, then α is an influenceable. If A, B and C are all controllables, then α is also a controllable. Uncontrollables are divided into the *observables* and the *unobservables*. The class of unobservable uncontrollables is not useful for deliberation since the agent cannot possibly know whether these are true or not.

Intuitively, an agent should only pursue the truth of propositions over which he has influence or control. But an agent should also take into account the truth values of the uninfluenceables and their logical consequences when determining his goals. To incorporate this kind of behaviour into QDT, Boutilier defines *complete knowledge goals* (Definition 2.16). *UI(KB)* is the subset of the default closure of a knowledge base on which an agent has no influence (Definition 2.15). A fact worth noting is that the worlds where *UI(KB)* is fully satisfied are the worlds that fill the bottom, most likely part of the normality ordering. A complete knowledge goal is a proposition that is true in the most ideal worlds where all elements of *UI(KB)* are true. No other worlds have to be considered by the agent, because they are not genuine possibilities. (Not all elements of *UI(KB)* are true in those other worlds.) Using the deductively closed set of uninfluenceables, an agent can now deduce a *complete action set* $\mathcal{A}$ that guarantees the truth of each complete knowledge goal α (Definition 2.17). A complete action set is a complete truth assignment over the set of controllables. In other words, a complete action set specifies for each possible action whether the agent should execute it or not. The agent should minimize the number of true propositions in $\mathcal{A}$, because actions have costs associated with them. The order in which the actions in $\mathcal{A}$ have to be executed is not specified. In most domains, the order of execution is important however, so Boutilier's architecture should be extended to make ordering actions in time possible.

Def 2.15 $UI(KB) = \{\alpha \in Cl(KB) : \alpha$ is uninfluenceable$\}$

Def 2.16 α is a complete knowledge goal iff $M \models I(\alpha|UI(KB))$ and α is controllable

Def 2.17 an atomic goal set $\mathcal{A}$ is a set of controllables such that $M \models UI(KB) \wedge \mathcal{A} \rightarrow \alpha$

Preference statements can now be interpreted differently. Boutilier gives the example "I prefer an umbrella when it is raining" which should now be interpreted as $I(U|UI(\{R\}))$ or "Ideally, I have an umbrella with me given the uninfluenceables in my knowledge base." In this case, *UI(KB)* is the singleton *{R}*, the only uninfluenceable and at the same time the only element of the *KB {R}*.

A QDT-agent determines reasonable courses of actions by first deducing all default consequences of its knowledge base and then reasoning with this knowledge as if it were certain. So a certain priority is given to defaults over preferences. Boutilier's algorithm first

determines all default consequences of the knowledge base. The extended knowledge base we end up with is then used for reasoning, along with the available preference information.

Boutilier points out that in order to create a truly rational agent, we need to calibrate the preference and normality orderings somehow. An agent has to consider both probability and utility measures when calculating the goal state. If a highly preferable state is also very unlikely to occur (e.g. due to the high probability of failure of an action), it is probably not worthwhile to pursue it. This idea is ancient; the most popular approach for doing this is arguably the maximization of expected utility [Savage 1954]. Boutilier proposes a method for calibrating preferences and normalities that works essentially the same as Savage's method, only it uses natural numbers instead of real values. In Section 4.3 we apply this idea to a worked example.

But what about domains with incomplete knowledge? Clearly, complete knowledge goals (Definition 2.16) are not appropriate here, because the agent does not have complete knowledge. If the agent does not know the outcomes of his actions and he does not have full accessibility to the world, surely the deliberation process described above would not work. Deliberation in such a non-deterministic environment has to consider several possible outcomes of a certain course of events (controllables, influenceables and uninfluenceables). The solution Boutilier proposes resembles the minimax approach from game theory (see for example [Russell and Norvig 1995]). He suggests that an agent should consider the worst result of each complete action set and execute the one with the *best worst* result. The minimax algorithm has been criticized to be sensitive to outliers in the following way. If an action has just one, highly unlikely but also highly unpreferable outcome, the action will not be chosen. Another action could have a better worst outcome, but also many other bad outcomes. Nevertheless, minimax will choose the one with the best worst outcome. Several refinements to minimax have been proposed so that it can handle such cases as well. QDT's deliberation algorithm could be refined in the same way. Boutilier agrees with this observation: He states that the worst possible outcome need not always occur, which leads to far from optimal solutions in some cases.

In order to say something about the transitive, connected preference relation on complete action sets, the notion of *action set dominance* of action set $\mathcal{A}_1$ over action set $\mathcal{A}_2$ is introduced. It is captured in Definition 2.18 ($\mathcal{A}_1 \leq \mathcal{A}_2$ or $\mathcal{A}_1$ is at least as good an action set as $\mathcal{A}_2$). It says that in at least one world cluster x on the preference ordering, the following three formulas hold: $\mathcal{A}_2$ is executed, every element of *UI(KB)* is true, making it a genuinely possible world[1] and $\neg\overleftarrow{\Diamond}_P(\mathcal{A}_1 \wedge UI(KB))$ is true. This last formula means that in no world cluster y less preferred than x, $\mathcal{A}_1$ and *UI(KB)* are true simultaneously. This in turn means that in such a cluster y, action set $\mathcal{A}_1$ should not be the executed set while all elements of *UI(KB)* are true (making it one of the most likely worlds). In other worlds, the worst possible outcome of $\mathcal{A}_1$ is not worse than the worst possible outcome of $\mathcal{A}_2$. This definition assumes implicitly that there is at least one outcome associated with the execution of $\mathcal{A}_1$. This assumption, together with Definition 2.18, implies that the worst outcome of $\mathcal{A}_1$ is better than the worst outcome of $\mathcal{A}_2$, the minimax idea. We claim that in most situations the method for maximizing expected utility described in Section 4.2 yields the same results as applying action set dominance, but it is less sensitive to the sort of outliers explained above.

[1] *UI(KB)* should be interpreted as the set itself or the conjunction of all its elements. In Definition 2.18, the latter interpretation should be used. Recall that possible worlds where all elements of *UI(KB)* are satisfied are at the bottom, most likely part of the normality ordering. So Definition 2.18 only tells us something about worlds that are likely to occur, which is good.

Def 2.18 $M \models \overleftarrow{\Diamond}_P(\mathcal{A}_2 \wedge UI(KB) \wedge \neg\overleftarrow{\Diamond}_P(\mathcal{A}_1 \wedge UI(KB)))$

Now, how to determine the best complete action set? To do this, we do not need to compare them all in a pairwise fashion, which reduces time complexity. Instead $M \models \mathcal{A}_i \leq \neg\mathcal{A}_i$ is the necessary and sufficient condition for "$\mathcal{A}_i$ is a best action set", because action sets are complete. The negation therefore means that at least one conjunct is false. This is actually the same condition as $M \models \mathcal{A}_i \leq \mathcal{A}_j$ *for each j*, but it is much easier (less expensive) to check. The cause for this convenience is that every complete action set $\mathcal{A}_j$ also satisfies $\neg\mathcal{A}_i$. When a different action set is chosen than the best, a worse outcome becomes possible (Definition 2.19). Intuitively, this means that the worst outcome of every action set $\mathcal{A}_j$ is worse than the worst outcome of $\mathcal{A}_i$.

Def 2.19 Let $\mathcal{A}_i$ be a best action set for KB and $\mathcal{A}_j$ be any complete action set. For any $w \models UI(KB) \wedge \mathcal{A}_i$ there is some $v \models UI(KB) \wedge \mathcal{A}_j$ such that $w \leq_P v$.

Next, Boutilier defines the necessary and sufficient condition for a *cautious goal* α: $\bigvee\{\mathcal{A}_i : \mathcal{A}_i \textit{ is a best action set}\} \models \alpha$. The disjunction of all best action sets should entail α. If $A \wedge B$ and $A \wedge \neg B$ are best action sets, then A is a cautious goal, but B is not. With $(A \wedge B) \vee (A \wedge \neg B)$ as a premise, it is easy to see that one can deduce A, but not B, either through proof by cases or by transforming the premise to conjunctive normal form.

In Figure 2.2, the most preferred world is $\{\neg D, \neg R, \neg U\}$, I walk to work without an umbrella and it is not raining. Since R is uncontrollable and unobservable, the agent should not adopt $\{\neg D, \neg R, \neg U\}$ as the goal world. The next best world is $\{D, R, \neg U\}$. In this example there is only one best action set $\mathcal{A}_i$: $\{D, \neg U\}$. So the necessary and sufficient condition for being a cautious goal ($\bigvee\{\mathcal{A}_i : \mathcal{A}_i \textit{ is a best action set}\} \models \alpha$) becomes $D \wedge \neg U \models \neg U$, which is true.

Finally, Boutilier applies the notion of *information value* to QDT. This is a useful concept in domains where retrieving information is costly and agents should only retrieve high valued information, that is likely to influence their decisions. An atom O has value if a best action set $\mathcal{A}_i$ with respect to KB is not a best action set for either $KB \cup \{O\}$ or $KB \cup \{\neg O\}$. In this case, retrieving information about the truth of O could lead to goal reconsideration. In all other cases, the effect of the truth of O on the goal state would be trivial and not worth retrieving.

After detailed examination of [Boutilier 1994], we arrive at the conclusion that QDT has a firm theoretical basis and substantial practical value. We now turn to some other approaches to qualitative decision theory. In particular, we devote considerable attention to Rao and Georgeff's BDI architecture.

2.3.5 Knowledge based systems (KBS)

Another approach to qualitative decision theory distinguished by [Dastani et al. 2003] is KBS or knowledge based systems. One of the most influential researchers of that approach is Judea Pearl [Tan and Pearl 1994]. He uses a calculus that is more quantitative in nature than QDT. It is therefore also referred to as a semi-qualitative method. We hardly use KBS as inspiration for our own architecture, so it is not covered here. We do mention it, because it forms an important part of the field of qualitative decision theory. It has been argued that KBS has greater expressive power than Boutilier's QDT [Dastani 2003]. However, it is also a more complicated and less intuitive approach. An interesting difference between Boutilier's theory

and Pearl's theory is that Pearl uses a bipolar scale for preferences. Worlds can be good, bad or neutral. Neutral worlds being represented by a preference value of zero, good worlds by positive numbers and bad worlds by negative numbers. We incorporate this idea into our own architecture.

2.4 BDI logic

Besides [Boutilier 1994], the theory presented in [Rao and Georgeff 1991] is another factor that influences our work. Their BDI architecture focuses on the generation of intentions. Observation, communication and learning lead to beliefs about the environment and other agents. These beliefs serve to constrain the set of goals (desires) the agent can have. The set of goals the agent ends up with is subjected to a reasoning step that extracts a consistent subset of the goal set and turns it into the set of intentions. These intentions should be viewed as the goals (desires) the agent will try to achieve in the current situation. Once such a set of intentions is found, the agent can execute the appropriate actions to reach a world that satisfies them.

The binary nature of the BDI architecture is the most striking difference with Boutilier's QDT. This causes the preferability of different intention-accessible worlds to be indistinguishable. An agent has an intention toward φ in the current world if and only if φ is true in every intention-accessible world associated with the current world. The agent can choose one of these worlds at random, they make equally good goal states. In QDT there is (in most cases) one cluster of worlds that has the best preference-utility-combination (Section 2.3), so the agent does not have to choose the goal state indeterministically. The winning cluster can still consist of many worlds, but this is due to incomplete knowledge. The worlds are the same with respect to what is known. In BDI, the different intention-accessible worlds can be fundamentally different, so BDI agents are often facing difficult choices. However, this corresponds to reality; humans also have to choose between different plans that are considered to be equally good to reach an intention φ.

QDT forms the basis for our model and we use the BDI architecture to incorporate the notion of time into the QDT approach. See Sections 2.5 and 4.1 for a detailed description of how QDT and BDI are combined. We do not need the distinction between goals and intentions from BDI, because we have Boutilier's preference ordering which serves more or less the same function. In fact, preference orderings make it possible to distinguish many levels of preferability, instead of just two.

Rao and Georgeff first describe their theory informally and then the syntax and semantics are explained formally. A BDI model is a collection of possible worlds where each possible world is a time tree with a single past and a branching future. A node in such a time tree is called a *situation*. These possible worlds are connected by three types of relations: *belief-accessibility*, *goal-accessibility* and *intention-accessibility*. The belief-accessibility relation connects a world to each world consistent with the agent's beliefs. The goal-accessibility relation connects a world to each world that the agent desires to be in, not considering the other knowledge he has. The intention-accessibility relation connects a world to each world the agent would actually try to achieve from it, taking into account all other knowledge it has.

2.4.1 Simple BDI example

To get a feel for the BDI architecture, we have examined a simple coin-toss game in detail; see Figure 2.3. The goal of the game is to guess the outcomes of two consecutive coin tosses. Player A tosses a coin and looks at it while making sure that player B does not see the outcome of the toss. Next, player B does the same. Then, both players guess the outcome of both tosses. This is of course not a fun game to play: Player A simply knows the outcome of the first toss and player B knows the outcome of the second toss. They just have to guess the other outcome to win.

One could distinguish three time steps in this game. It starts from a single possible world, the one on the left of Figure 2.3. There are always two possible events from one point in time to the next, resulting in four possible worlds at the end of the game. Let us consider the situation where both tosses turn out to be heads, the actual world at the end of the game is the one in the top-right corner of Figure 2.3. At $t = 0$, neither player knows anything about the outcomes. At $t = 1$, player A knows the outcome of the first toss and player B is still in the dark. Thus, player A has one belief-accessible world from $t = 0$, whereas player B has two. At $t = 2$, player B knows the outcome of the second toss, but player A does not. So player A has two belief-accessible worlds from his single possible world at $t = 1$, resulting in two possible worlds at $t = 2$: <*heads, heads*> and <*heads, tails*>. Player B has one belief-accessible world from each of his two possible worlds at $t = 1$, also resulting in two possible worlds at $t = 2$: <*heads, heads*> and <*tails, heads*>. Note that the number of branches in a possible world declines when more information becomes available about the events that have occurred. If the players would intersect their belief set by communicating with each other, they would discover that the only possibility is <*heads, heads*>. On their own however, they have to guess one of the outcomes.

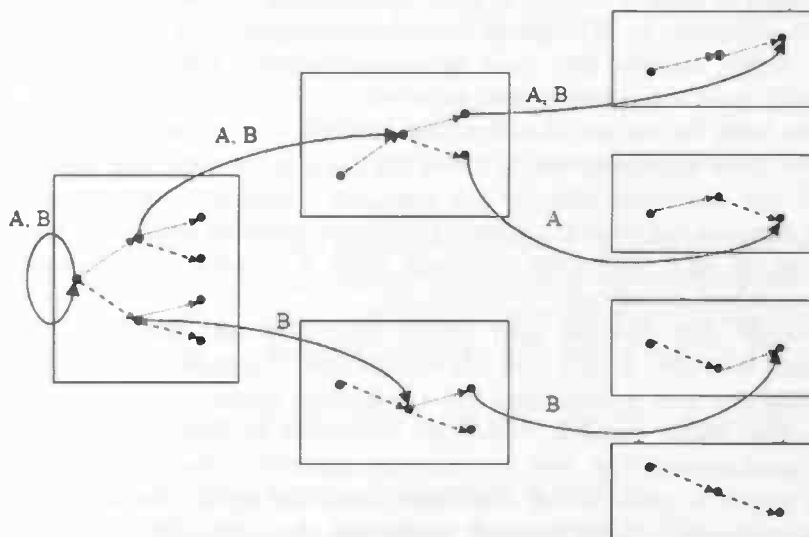


Figure 2.3: BDI model for the coin-toss example. Player A tosses the first coin, player B tosses the second coin. Dotted lines represent heads-toss-events, dashed lines represent tails-toss-events and solid lines represent belief-accessibility relations. The A's and B's above the solid lines indicate for which player the belief relationship holds.

In this game, we have only belief-accessibility relations between worlds. It does not make sense to talk about goal-accessible relations (indicating which worlds an agent desires to reach) or intention-accessible relations (indicating which worlds an agent will actually try to reach given all his other knowledge). But these other relations work just like belief-accessibility relations.

2.4.2 Formal definitions of BDI's logic and model

In a rational agent, a goal-accessible world is always a sub-world of a belief-accessible world, making sure that he does not desire to achieve a world that he believes impossible to reach. This is roughly the same as saying that the agent employs the rule GOAL(φ) $\rightarrow$ BEL(φ). Being a sub-world means that the time-frame[1] of each goal-accessible world has to be a sub-tree of a belief-accessible world. Furthermore, the valuations of the propositions in each goal-accessible world have to be a subset of the valuations in a belief-accessible world. In the same way, a rational agent employs the rule INTEND(φ) $\rightarrow$ GOAL(φ). This is more or less equivalent to the fact that each intention-accessible world is a sub-world of a goal-accessible world. This constraint makes sure that the agent only pursues worlds it desires to achieve.

The language Rao and Georgeff use is based on standard first order logic with a finite set of atoms and the usual connectives. They use A and B for atomic sentences and φ_1 and φ_2 to represent formulas. In their extended multimodal predicate logic, they allow two distinct types of formulas: state formulas and path formulas. Formulas of the former type are evaluated at a certain situation (time point in a given world) whereas the latter type is true or false along a certain path in a possible world (time tree). The single-agent architecture can easily be extended to a multiagent architecture using a subscript for each agent's attitudes. Rao and Georgeff's inductive definitions of state formulas and path formulas are given in Definitions 2.19 and 2.20 respectively.

Def 2.19 State formulas

- First order formulas are state formulas.
- If φ_1 and φ_2 are state formulas and x is an individual or event variable, then: $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$ and $\exists x \varphi_1(x)$ are state formulas[2].
- If e is an event type, then *succeeds(e)*, *fails(e)*, *does(e)*, *succeeded(e)*, *failed(e)* and *done(e)* are state formulas.
- If φ is a state formula, then BEL(φ), GOAL(φ) and INTEND(φ) are state formulas.
- If φ is a path formula, then *optional*(φ) is a state formula.

Def 2.20 Path formulas

- State formulas are path formulas.
- If φ_1 and φ_2 are path formulas, then: $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $\varphi_1 \cup \varphi_2$, $\Diamond\varphi_1$ and $\circ\varphi_1$ are path formulas.

Events (e) are associated with transitions between situations. In BDI we have a branching future, because at a certain point in time, several events could occur. Each possibility results in an extra branch (see Figure 2.3). In QDT, events can be controllable, influencable or

[1] The points in time within the worlds (situations) with $\prec$ relations between them, but no information about the truth values of propositions.

[2] This definition is minimalistic in the sense that it does not include the basic symbols for conjunction, implication, bi-implication and universal quantification; these can of course be defined in terms of the symbols for disjunction, negation and existential quantification.

uncontrollable. In BDI, events are always actions by the agent himself and branches in time-trees are therefore choices for the agent[1]. Rao and Georgeff distinguish between *primitive events* and *non-primitive events*. Primitive events are actions directly performable by an agent, mapping to an adjacent point in time. Non-primitive events map to non-adjacent points in time. These allow the designer to model the partial nature of plans.

An agent's actions can fail or succeed. *succeeds(e)* is true if event e has the expected effect when executed in the next time-step, *fails(e)* is true when e does not have the expected effect when executed in the next time-step and *does(e)* is true when e is executed the next time step, regardless of its outcome. *succeeded(e)*, *failed(e)* and *done(e)* have the same meaning, but then for the previous time-step. The formulas BEL(φ), GOAL(φ) and INTEND(φ) mean that the agent believes φ is true, has the goal to make φ true and has the intention to make φ true respectively. BEL(φ) is true in a situation if and only if φ is true in every belief-accessible world of that situation. The same holds for GOAL(φ) and INTEND(φ) and goal-accessible worlds and intention-accessible worlds respectively. Finally, *optional(φ)* is true at a point in time if path formula φ is true along at least one of the paths emanating from this point and *inevitable(φ)* is true if path formula φ is true along every path emanating from this point (*inevitable(φ)* $\equiv_{df}$ $\neg$*optional($\neg\varphi$)*).

State formulas are also path formulas. If a state formula is true at every point on a certain path, it is true as a path formula along that path. Until-formulas can express path formulas of the form "φ_1 will be true until φ_2 becomes true" ($\varphi_1 \cup \varphi_2$). A statement of the form $\Diamond\varphi_1$ expresses the fact that φ_1 will eventually become true. The formula $\circ\varphi_1$ says that φ_1 will be true at the next point in time and $\Box\varphi_1$ means that φ_1 will be true from now on.

Def 2.21 *an interpretation* $M = <W, E, T, \prec, U, \mathcal{B}, \mathcal{G}, \mathcal{I}, V>$

W	set of worlds
E	set of primitive event types
T	set of time points
$\prec$	binary relation on time points
U	universe of discourse
$\mathcal{B}$	mapping of current situation (w_t) to its belief-accessible worlds more formally: $\mathcal{B} \subseteq W \times T \times W$
$\mathcal{G}$	maps current situation to its goal-accessible worlds (works like $\mathcal{B}$)
$\mathcal{I}$	maps current situation to its intention-accessible worlds (works like $\mathcal{B}$)
V	maps elements in U to truth values for any situation w_t
w_t	situation; a world $w \in W$ at time point $t \in T$

Def. 2.22 *A world (time tree)* $w \in W$ *is a tuple* $<\mathcal{T}_w, \mathcal{A}_w, \mathcal{S}_w, \mathcal{F}_w>$

$\mathcal{T}_w \subseteq T$	the set of time points in world w
$\mathcal{A}_w \subseteq \prec$	only defined for the members of $\mathcal{T}_w$
$\mathcal{S}_w$	function mapping pairs of adjacent time points to primitive events more formally: $\mathcal{S}_w : \mathcal{T}_w \times \mathcal{T}_w \mapsto E$ $\mathcal{S}_w$ tells the agent in what situation he will be if event e is successful
$\mathcal{F}_w$	works like $\mathcal{S}_w$ $\mathcal{F}_w$ tells the agent in what situation he will be if event e is not successful

[1] In the example depicted in Figure 2.3, we did not obey this constraint however.

Definitions 2.21 and 2.22 define part of the semantics for the BDI architecture. We refer to [Rao and Georgeff 1991] for the entire description of the semantics. The part of BDI that is particularly useful to our project is the way it handles time series and events.

2.4.3 The BOID architecture

Another popular approach to reasoning about mental attitudes is the BOID architecture [Broersen et al. 2002]. We treat this study, because BOID is relatively easy to comprehend and it stands closer to practical application than BDI[1]. In the BOID archtecture, intentions are generated from the interaction between beliefs, obligations and desires. It therefore has much in common with the BDI architecture (BOID is BDI plus obligations). These mental attitudes were chosen for their intuitive nature and the useful agent types they result in. Both BDI and BOID use binary representations and they handle conflicting knowledge in more or less the same way. However, the BDI architecture uses a monotonic logic where mental attitudes are not conditional in nature. The BOID architecture considers mental attitudes to be conditional and depending on context. It uses a non-monotonic logic based on Reiter's default logic [Reiter 1980] (much like QDT). Both BDI and BOID take into account the side effects of an action. If an agent desires to go to the beach and if a side-effect of going to the beach would be losing one's job, he will not actually intend to go to the beach.

Jan Broersen and his colleagues use a strong overriding mechanism. This mechanism is helpful for an agent because it will prune the search space during deliberation. For example, an agent should have his beliefs override his desires, making sure that he will not desire things that seem impossible to achieve. The order in which the four mental attitudes (beliefs, obligations, intentions and desires) override each other determines the type of the agent. One of the more useful agent types is the social realistic agent. In such an agent, obligations override desires (social) and beliefs override desires (realistic). We refer to [Broersen et al. 2002] for an overview of the different agent types in the BOID architecture.

The objective for BDI agents and BOID agents alike is to determine appropriate intentions. Beliefs are generated while observing the world as well as communicating and learning. These are then used to make sure that no desires and obligations are adopted that are inconsistent with what the agent believes. At this point the agent has a set of desires and obligations. These can be inconsistent, so the agent has to determine a consistent subset of this desire/obligation set and turn it into the set of intentions. Obligations are an interesting extension for at least two reasons. First of all, obligation is a natural mental attitude within a group of agents. The term obligation is often associated with social norms. Furthermore, the distinction between desires and an obligations makes it possible to consider one of them to be superior to the other, yielding selfish or social agents.

Figure 2.1 is the graphical equivalent of Figure 2.4 (taken from [Broersen et al. 2002]). First, a priority function ρ is selected. Then the agent enters an infinite loop. One iteration through the infinite loop proceeds as follows. The agent stores his observations in *Obs*. Together with the agent's beliefs, obligations, intentions, desires and priority function, it will generate the candidate goal sets (default *extensions*) and store them in *S*. Then *S* is fed to the module that selects a good goal set (bounded rationality) and generates a set of plans to satisfy the formulas in this goal set. Finally, all information is updated and another iteration begins.

---

[1] The BDI architecture has been implemented successfully several times, but the architecture itself is extremely theoretical in nature. One might say that the gap between theory and practice is wider than with BOID.

```
select ρ;
repeat
  Obs := read_environment;
  S := generate_candidate_goal_sets(Obs, B, O, I, D, ρ);
  P := select_goal_set_and_generate_plans(S);
  update(B, O, I, D, ρ, P);
until forever
```

Figure 2.4: Deliberation and means-end reasoning loop in BOID at a high level of description.

Broersen et al. use a language with a finite number of propositions and an agent's mental attitudes are conditionals (just like in QDT). Arguably this is a pro for the BOID architecture. In [Broersen et al. 2002] technical details are kept to a minimum. For this paper, they made the assumption that extensions are sets of literals. No complex formulas are allowed.

Mental attitudes are maintained in a set of propositions of the form $a \leftarrow X \rightarrow b$ with $X \in \{B, O, I, D\}$ meaning that if a is true, the agent has mental attitude X toward b. More precisely, in the case of obligation for example, $a \leftarrow O \rightarrow b$ means that if the agent has a as a goal, it feels obliged to also adopt b as a goal. The priority of the four different types of rules depends on the type of the agent. $a \leftarrow X \rightarrow b$ is not a solid implication. Instead, it states that a is a necessary, but not sufficient condition for b. Other rules can prevent the rule from firing.

Default logic [Reiter 1980] takes care of this non-monotonic reasoning by deriving extensions. This reasoning mechanism poses a consistency constraint on the modus ponens rule. Furthermore it introduces non-determinism into the BOID architecture. The order in which the applicable default rules fire can be chosen at random or by a priority function ρ. In practice, not all extensions are calculated. When the agent has used a certain amount of resources, it will stop and return the best result it has come up with so far (bounded rationality [Simon 1987]).

In BOID, *prioritized* default logic is used: The defaults with the highest priority fire first. The priority function ρ maps every rule in the set $B \cup O \cup I \cup D$ to an integer, indicating its priority. In different agents, the four different types of rules can have different values assigned to them, leading to differences in behaviour. In a social realistic agent for example, beliefs always have a higher priority than obligations and obligations always have a higher priority than desires: $\rho(a \leftarrow B \rightarrow b) > \rho(c \leftarrow O \rightarrow d)$ and $\rho(e \leftarrow O \rightarrow f) > \rho(g \leftarrow D \rightarrow h)$ for every a, b, c, d, e, f, g and h. In our opinion, this is not the best way of modelling a social realistic agent, we think it is too extreme. Even an agent that is highly social should still prioritize his most important desires over his least important obligations. Fortunately, ρ can be specified any way we want, so this behaviour would not be hard to achieve in the BOID architecture. Goal generation in BOID is only efficient if every rule has a unique priority value. How to optimize the procedure for the more general case is still an open problem. Finally, it is easy to see how ρ can facilitate learning. In QDT, shifts in the priority and normality ordering as well as the injection of new information into them change the agent's behaviour. In BOID, reassigning rule-priority leads to adaptive behaviour.

In the goal selection procedure, the extensions are chosen at random or subjected to a priority function, just like the default rules are prioritized in the goal generation procedure. This gives rise to even more different types of agents than we already had from the priority order of $B \cup O \cup I \cup D$. An agent could be persistent in the sense that it has a preference for extensions that show resemblance with the extensions that he chose before. He could also be of the conservative type, meaning that he will choose an extension that looks like the current state of affairs. The conservative type is also an economic one, because little changes usually means low costs.

If an agent cannot construct a feasible plan to reach the chosen extension, he chooses a different extension to pursue. The preconditions and effects of actions and action sequences are stored in belief-rules. Actions that have made it through the planning module are stored as intentions. Recall from Section 2.3 that Boutilier distinguishes between controllables, influencables and uncontrollables. Broersen and his colleagues make more or less the same distinction. Furthermore, an agent should take into account the side-effects of his actions. After observation, goal generation and goal selection, the agent's mental attitudes are updated. The changes that are made, alter the agent's behaviour. [Broersen et al. 2002] contains a short but impressive description of the dynamic capabilities of the BOID architecture.

2.4.4 Collective intentions

We conclude Section 2.4 with a short review of another study that extends the BDI architecture: [Dunin-Kęplicz and Verbrugge 2002]. We treat this study, because it focuses on teams of agents, whereas QDT, BDI and BOID are mostly concerned with the behaviour of a single agent. Barbara Dunin-Kęplicz and Rineke Verbrugge define a team as a group of agents that have a common intended goal which they achieve by cooperating and assisting each other. They also characterize formally what it means for a team to have a collective intention. There is more to it than just having the same intentions. For sake of simplicity, they leave out the notion of time. Their full theory does include time however. It can handle either a linear time frame like in [Cohen and Levesque 1990] or a branching time frame like in [Rao and Georgeff 1991].

In this study, rationality rather than psychological soundness is pursued. There is a common agreement that intentions play an important role in rational decision making: They drive means-end reasoning, they constrain future deliberation and they influence future behaviour. [Rao and Georgeff 1991] presents us with a theory for rational BDI-agent behaviour. However, in case of a team of agents with collective intentions, things can get much more complicated. [Dunin-Kęplicz and Verbrugge 2002] and its companion papers extend the BDI-architecture with ways to handle this extra level of complexity. This concludes our discussion of the literature on qualitative decision theory. We now turn to the combination of QDT and BDI that we need for our own architecture.

2.5 Combining QDT and BDI

[Dastani et al. 2003] provides a comprehensive overview and comparison of the different approaches to automatic decision making. Dastani and his colleagues start with a short description of modern Bayesian decision theory or CDT. This description is similar to that of [Doyle and Thomason 1999], which has already been treated in Section 2.1. Next, Dastani et al. compare CDT to QDT. The main difference between the two approaches is that CDT uses quantitative probabilities and utilities (real values between 0 and 1 for probabilities and between -1 and 1 for utilities) where QDT uses normality orderings and preference orderings on possible worlds.

So CDT uses purely quantitative representations where QDT uses orderings on clusters of possible worlds. BDI systems take the abstraction to the extreme, using binary values to represent likelihood and preference. The consequence of this abstraction is that an agent either believes something or not and it wants to make something true or not. There are no gradations in between like in QDT. We are particularly interested in Rao and Georgeff's combination of epistemic logic and time series using two sets of modal operators.

For our own architecture, QDT is combined with the notion of time used in BDI. We call this new architecture QDT+. The combined model we end up with is both general purpose and suitable for an adaptive diary assistant. We choose QDT as the basis for our model, because it can reason about different levels of preferability and normality. BDI represents beliefs, desires and intentions in a binary way. It lacks the power to distinguish between more than two levels.

Our model consists of three modalities: P for preference, N for normality and T for time. Information about an agent's preferences and normalities is represented in formulas of the form $I(B|A)$ and $A \Rightarrow B$ respectively as defined in Definition 2.7. We can also compare the preference or normality of two propositions directly using formulas of the form $A \leq_P B$ and $A \leq_N B$. These formulas are interpreted using Definition 2.8. The corresponding semantics are linear orderings of clusters of possible worlds. These two modalities work exactly the same, the only difference between them is how their meaning is interpreted. We adopt Pearl's idea to use a bipolar ordering for preferences. Worlds can be relatively good to different degrees, relatively bad to different degrees or neutral. A bipolar ordering for normalities does not make sense.

Craig Boutilier's QDT theory gives reasoning about normalities a certain priority over reasoning about preferences. First, the knowledge base is extended with its default consequences. Then, the default closure of the knowledge base is used in the next reasoning step, along with information about preferences. We agree with this order, therefore, we adopt it in QDT+. An agent should first consider the uncontrollables before looking at the controllables and influencables. That normality rules have priority over preferences can also be seen from Definition 2.18.

Information about the third modality, time, is represented in formulas of the form $A \leq_T B$ as in Definition 2.8. It does not make sense to also introduce formulas for time of the form described in Definition 2.7. The semantics for time-series are trees of possible worlds that branch into the future. Reasoning about time is practically identical to the way it is done in BDI. The most important difference is that we use one branching time-tree of possible worlds where Rao and Georgeff use a set of possible worlds with belief, desire and intention-accessible world relations between them, giving rise to a network of possible worlds. Each of these worlds is a branching time-tree in itself with information about which propositions are true at the nodes. We do not need this trees-in-a-network representation of BDI, because beliefs, desires and intentions are represented in an altogether different way in our QDT+ architecture; as linear preference and normality orderings. In our QDT+ model, the information that is stored in the branching time-tree is only used during learning. An agent can predict the future by looking at the past. He derives new rules, leading to changes in the preference and normality ordering.

The modality time serves to store the past states of the environment and the agents. It is not to be confused with the way we represent dates and times in the diary. This is done using first order predicates and should therefore not be interpreted as a modality at all. QDT uses propositional logic. We use predicate logic (like BDI), because it forms an integral part of the OutOfBrain language. Unconstrained predicate logic is fundamentally undecidable. However, the use of a uniquely indexed database combined with a finite domain rectifies the use of unconstrained predicate logic[1]. In combination with arithmetic, it is well suited for representing times and dates in a diary system. Effectively, we are integrating quantitative information into a qualitative framework here. This choice brings along some extra complexity, especially when the number of values that variables can bind to is large. For some thoughts on search complexity, see Section 3.1.7.

This concludes our treatment of the literature on qualitative decision theory. The next Chapter is concerned with literature on graph rewriting and other topics.

[1] Constrained predicate logics are sublanguages of unconstrained predicate logic. A popular constrained predicate logic is description logic [Nardi and Brachman 2002]. In this language, only unary and binary predicates are allowed. It is used for the semantic web in combination with OWL [Baader et al. 2003].

3 Literature on graph rewriting and other topics

Our secondary focus while studying the literature is on existing graph rewriting techniques and how these relate to the OutOfBrain system (Section 3.1). The remainder of the current Chapter covers some other interesting topics from literature. These topics however, are only covered briefly, so as not to distract us from our main goals. Section 3.2 is about communication between agents and how it aids in the construction of a cooperative team. Section 3.3 provides some insight in the user modelling techniques at our disposal. Some thoughts on learning in a qualitative decision architecture have already been dealt with in Chapter 2, because in most cases, such an architecture is designed bearing the capability of learning in mind. Finally, Section 3.4 minimally covers interface design. Cognitive ergonomics is a whole different ballgame and we cannot get into it in detail in this Master's thesis. However, we feel that the field should have some influence on every software-design project.

3.1 Graph rewriting and OutOfBrain

Graph representations are used extensively in modelling techniques. The most popular example of graph representations is arguably the universal modelling language (UML) [Rumbaugh et al. 1998]. UML is the result of an enormous combined effort by a variety of companies and research institutes. In 1997, version 1.1 of the modelling language was released, resulting in a convenient standardization in the field of software design. It is also used in other domains, most notably in hardware design, modelling business processes, engineering, and organizational structures. An UML model can provide a team of developers with a common frame of reference. Another popular type of graphical representations, one that is especially suited for artificial intelligence, is the semantic network.

Graph representations are less popular at the implementation level for several reasons. We will explain these reasons shortly. Despite this unpopularity, the graph rewriting community does not give up that easily. It is only a matter of time and effort before a graph rewriting system for implementation will emerge that can do most of the things a textual programming environment can do. We expect that such a graph rewriting system will be especially successful in hybrid settings. Some functionality cannot be implemented in graphs conveniently, so we would have to rely on textual programming languages there. OutOfBrain, currently under development at Rockingstone, will be such a system. One of this project's goals is to test and improve the current version of OutOfBrain during the development of an adaptive diary assistant.

Section 3.1.1 introduces graph rewriting as a programming language and Sections 3.1.2 and 3.1.3 summarize the different approaches in the field. There are many possibilities and it is not clear which approach is the best one. It also depends of course on the task at hand. Sections 3.1.4 and 3.1.5 introduce another type of graphical programming called *coloured Petri nets*. In Section 3.1.6, we present the basics of OutOfBrain so that we can compare it to other graph rewriting techniques and coloured Petri nets in Section 3.1.7. Finally, we devote some attention to the formal properties of OutOfBrain, also in Section 3.1.7. In order to do this, we borrow some results from literature on graph rewriting techniques that resemble OutOfBrain.

3.1.1 Introduction to graph rewriting

A graph rewriting system consists of a host graph and a set of rewrite rules. In the context of multiagent systems, the host graph is a representation of the world, whereas the rewrite rules are descriptions of how the world can change due to actions by the agents and other events. A comprehensible way of looking at these graphs is as sets of logical formulas. The host graph corresponds to a set of logical facts and the graph rewrite rules are the implications that operate on these facts.

Terminology is far from standardized in the field of graph rewriting. A clear terminology is given in [Blostein et al. 1995]. It is represented in a graphical example in Figure 3.1. There are many variations in graph rewriting. However, they all follow the same basic mechanism. A control module picks rewrite rules one by one, either at random or according to some ordering function. Once a rule has been chosen, the control module will search the host graph for a matching subgraph. If it finds one, the rule will fire as shown in Figure 3.1. The rule $g_l \rightarrow g_r$ replaces all occurrences of g_l in the host graph with g_r. Subgraph g_l^{host} is isomorphic to g_l, so it is transformed into g_r^{host}, which is isomorphic to g_r. What it means exactly for two (sub)graphs to be isomorphic depends on which graph rewriting variation is used. We will get into these differences in the next Section. The edges marked with asterisks are called pre-embedding edges and the ones that are marked with hash symbols are called post-embedding edges. Note that in Figure 3.1 there is one post-embedding edge for every pre-embedding edge, but this is not generally the case. We now turn to a classification of existing graph rewriting techniques.

3.1.2 Variations in graph rewriting mechanisms

The most obvious way of classifying graph rewriting mechanisms is by analysing which types of graphs are allowed. The nodes in a graph can have labels or not. The same holds for the edges between the nodes. Furthermore, in most graph representations directed edges are used, but undirected edges are also possible. With undirected edges, the system cannot distinguish between for example *R(a, b)* and *R(b, a)*. Some graph rewriting mechanisms allow for node attributes. These can be data fields of any complex type. All these variations are universal machines, but it should be clear that graphs without labels or node attributes cannot be used in practice for complex modelling. The distinction between simple and complex graph rewrite mechanisms resembles the distinction between machine language and high level programming languages.

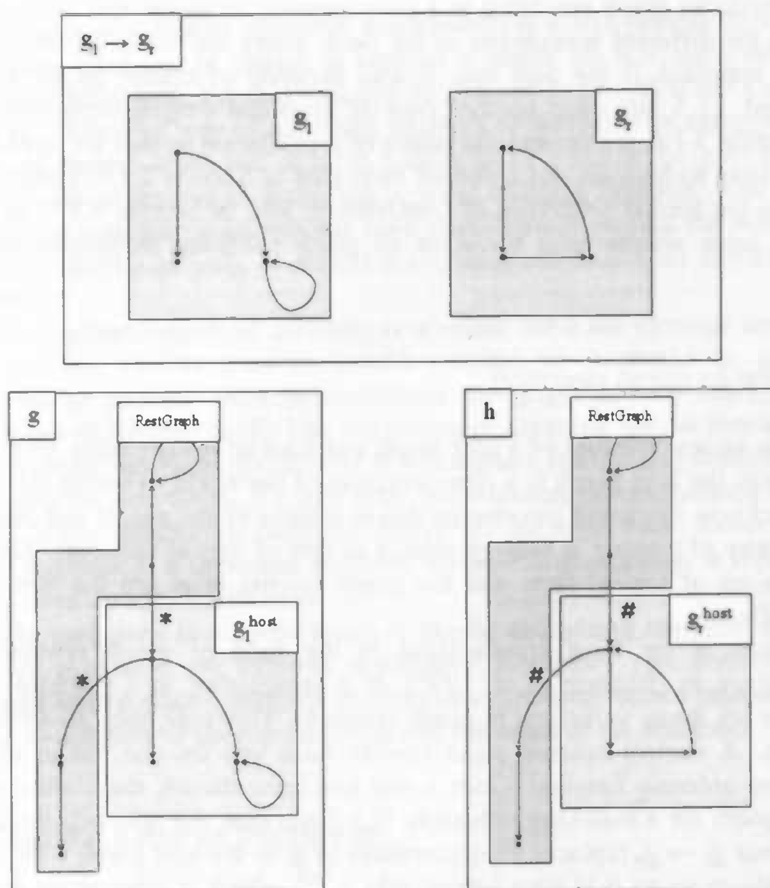


Figure 3.1: Graphical representation of the graph rewriting ontology used in [Blostein et al. 1995]. The graph rewrite rule $g_l \rightarrow g_r$ replaces all occurrences of g_l in the host graph with g_r. Graph g is an appropriate host graph, because g_l^{host} is isomorphic to g_l. Graph h is the corresponding result graph, because g_r^{host} is isomorphic to g_r. Asterisks in g indicate pre-embedding edges, hashes in h indicate post-embedding edges.

In [Blostein et al. 1995] graph rewriting mechanisms are also classified according to their gluing method. Each graph rewrite rule has to provide embedding information, either explicitly or implicitly. When a rule matches g_l^{host}, a subgraph of the host graph isomorphic to g_l, is removed (Figure 3.1). This leads to *dangling* edges. These pre-embedding edges (that used to connect g_l^{host} to RestGraph) are now dangling in space, having lost their from-node or to-node[1]. These edges have to be connected to the correct nodes in g_r^{host} (the substitution for the removed subgraph g_l^{host}). This process leads to new connections called post-embedding edges.

The form of graph rewriting depicted in Figure 3.1 uses an implicit gluing method. It derives implicitly which nodes in g_r^{host} have to inherit connections from the nodes in g_l^{host}. It does this by analysing the shape of both subgraphs. A node in g_l^{host} with the same relative position as a node in g_r^{host} inherits all connections from it. This is why the graph rewriting mechanism in Figure 3.1 does not need explicit embedding information. Most graph rewriting mechanisms do need it however. Instead of looking at the physical locations of the nodes, these mechanisms only analyse what connections a node has and how the nodes and edges are

[1] Graph rewriting lingo for the node that the edge starts from and the node that the edge points to respectively.

labelled. Thus, the system needs no information whatsoever about the graphical layout of the graphs. The graphical representations are only of interest to the designer, i.e. they improve usability. The problem with implicit embedding, based on physical locations, is that the system has to be aware of the relative positions of the nodes as well as the connections they have with other nodes. This increases search times dramatically. Implicit embedding is therefore not the most popular approach. A disadvantage of explicit embedding though is the occurrence of unexpected matches. In Figure 3.1, if the system would not analyse the physical locations of the nodes, at least two more subgraphs of g would match the rule. This problem can be solved quite easily by adding labels to the nodes in order to distinguish them from one another. To allow for node and edge labels also increases search times, but we need the labels anyway for convenient representations of logic formulas. *Subgraph isomorphism* determines when a rule finds a match in a host graph in cases where embedding information is implicit, i.e. based on the physical locations of the nodes. But when embedding information is explicit, the term *general morphism* is used. These two morphisms can also be combined by indicating for each rule whether it should use subgraph isomorphism or general morphism.

Choosing an appropriate explicit embedding mechanism can involve a trade-off between ending up with many simple rules or just a few complex ones. If the specification of the embedding information is unrestricted, the graph rewrite rules can get highly complex, but only a few are needed. This makes sense, because a single rule can have high expressive power if no constrains are imposed on it. The host graph can be altered in a profound way by applying just one rule, where restricted embedding mechanisms would need the application of several rules. However, unrestricted methods lead to a high search complexity. Furthermore, the highly constrained gluing methods have the strongest mathematical basis. The key is to find the right balance between modelling power and formal properties like worst case time complexity, the existence of useful mathematical theorems and proofs about the integrity of a database. What follows is an overview of constrained gluing methods in increasing order of complexity, taken from [Blostein et al. 1995].

Constraints leading to better formal properties

The first constraint (leading to lower complexity and stronger mathematical basis) is *preservation of orientation and label*. For each post-embedding edge there has to be a pre-embedding edge that has the same orientation and label. This comes down to the fact that after the removal of g_l^{host} the dangling edges cannot be removed, they have to be reconnected to g_r^{host}. New edges can still be added though. The second constraint that can be posed on the gluing method (leading to lower complexity and stronger mathematical basis) is called *depth1*. Here, the post-embedding edges can only point to or start from the nodes in RestGraph that are direct neighbours of g_r^{host}. To know which nodes are the direct neighbours, the system has to be aware of the geographic locations of the nodes.

Constraints on the allowed gluing steps can easily be combined. For example, the constraint *simple* is preservation of orientation and label combined with depth1. Another combined constraint is called *elementary*. It combines simple with the additional constraint that the gluing cannot depend on the labels of nodes in RestGraph. The preconditions of the rewrite rules are not allowed to contain information about these labels. *Analogous* is the same as elementary, again with an extra constraint added. It says that the gluing cannot depend on the orientation and labels of pre-embedding edges. All that is left for specifying preconditions then, are the nodes in g_l^{host}, their labels, the edges and labels between them and the edges that connect them to RestGraph without orientation or label information. The last constraint presented here is called *invariant*. It says that there has to be exactly one post-embedding edge for each pre-embedding edge.

Another source of variation in rewrite mechanisms is which priority function they use on the rewrite rules. They can be chosen at random to check for rule applicability, reflecting non-deterministic behaviour, but they can also be ordered in several ways. Learning this ordering presents an interesting way of achieving adaptive behaviour. The designer can also determine what should happen when two or more subgraphs of g match the g_l of a rule. Should one of them be chosen at random or according to some ordering function? And what should happen when the first matching subgraph has been transformed? Should the rule transform the other matching subgraphs as well, or should the system continue with checking the next rule for applicability?

Yet another source of diversity is whether there are external application conditions. Graph rewriting can be used in hybrid systems, so it should be possible for external components to impose extra preconditions on graph rewrite rules, in addition to the preconditions of the rule itself. The last source of diversity we put forward here is the use of attribute transfer functions. Recall that in some graph rewriting systems, nodes are allowed to contain complex data fields. If a rule fires, this data can be manipulated. Such node attributes make for better structuring facilities. However, the expressive power does not change; a universal machine is a universal machine, whether complex data is represented in the node attributes or in the graph structures.

The variations in allowed types of graphs and gluing methods, the rule ordering and whether or not there are external application conditions and attribute transfer functions provides us with a wide variety of mechanisms. The best choice depends on the problem at hand, the availability of appropriate tools and the tastes of the designer.

3.1.3 Choosing a rewrite mechanism

Many factors have to be considered when choosing a rewrite mechanism for implementation. It should be clear that the invariant gluing method is inconvenient for many common rewrite operations. But surely, the unconstrained method would yield an unacceptable worst case time complexity and a profound lack of mathematical basis[1]. Generally, designers choose a mechanism that is somewhere in the middle. They find it easier to express transitions using more rules than strictly necessary, because it makes the system easier to understand and it yields better formal properties.

In theory, the graphical representations used in graph rewriting make for a good manageability for the programmer. In practice however, this is only true for small programs. Graph rewriting is not good at scaling up what it can do in simple situations. This is due to the simple fact that good tools for graph rewriting have not yet been developed. A serious attempt to create such tools is the environment PROGRES and of course the environment we work in: OutOfBrain. While these two projects make a good effort, much work remains to be done. An important feature that these and other tools are lacking is modularity. Object oriented programming has achieved an inviolable status in the world of computer science during the last two decades. Therefore it is clear (to us and to Dorothea Blostein and her colleagues) that the modular architecture of object oriented programming should also be incorporated into graph rewriting for it to be successful. Especially in large software projects, modularity and reusability are essential. The same holds for good editing tools. A programmer using graph rewriting should be able to work with several tabs, collapsing and expanding subgraphs,

[1] OutOfBrain does not pose many constraints on gluing. The reason this does not lead to complexity problems is the smart database that operates in the background. In this database, each node has a unique identifier besides its label. These identifiers speed up searching for matching subgraphs substantially.

search options, zooming facilities and much more[1]. Both PROGRES and OutOfBrain do not yet have all these features installed. The development of good tools for OutOfBrain will form an important part of Rockingstone's future work.

3.1.4 Coloured Petri nets

There is another graphical programming technique that deserves some attention. Carl Adam Petri is the father of the paradigm that is known as low-level Petri nets today. We treat some literature on Petri nets, because we want to develop a rich view of existing graphical programming techniques. Invented in 1962, Petri nets have had considerable influence on the field of parallel computing. Systems are described in diagrams of ellipses called *interface places* and circles called *internal places*. They correspond to the possible states of the system. Several of these can be active simultaneously. Possible actions are represented as rectangles called *transitions*. The arrows connecting these components are called *arcs*. Each arc is associated with an *arc expression*, which describes how the state of the system changes. A Petri net can be used to describe the synchronisation of concurrent processes.

[Jensen 1992] is the first volume of a book that covers many aspects of coloured Petri nets in depth, from theoretical foundations to practical use. Coloured Petri nets extend low-level Petri nets with the power of programming languages. In coloured Petri nets, each place has a set of *tokens* that can be of any complex data type, whereas in low-level Petri nets, the places contain no information whatsoever. Kurt Jensen compares coloured Petri nets with high level programming languages and low-level Petri nets with assembly code. They have the same expressive power, but in practice, coloured Petri nets are capable of modelling more complex systems, because they have better structuring facilities, like types and modules. The same comparison can be drawn between low-level Petri nets and graph rewriting without node attributes on the one hand and coloured Petri nets and graph rewriting with attributes on the other hand (see Section 3.1.2). However, where low-level Petri nets are considered to be inferior to coloured Petri nets, this does not hold for graph rewriting with and without node attributes. Fairly complex systems can be modelled in graph rewriting, even without node attributes. This is because graph rewriting mechanisms without node attributes still have node and edge labels to store information in.

First there were only a few "colours" allowed in coloured Petri nets, but the Petri net research community soon realised that data in the places could be of arbitrary complexity, even entire arrays of data with many different data types are allowed. It is due to this fact that coloured Petri nets are capable of modelling colossal industrial processes.

When executed, a coloured Petri net begins in some initial place and starts checking if any transitions match. In order to have a match, the place has to possess tokens that match the types of the variables in the arc expression that leads from the place to the transition. If so, the transition occurs and the data in the initial place moves (i.e. binds) to the variables in the arc expression and the transition is said to be *enabled*. From the transition rectangle, the algorithm will look for matching edges again. Some or all of the data is transferred to the arc expressions of an arc leaving the transition. This process continues, possibly forever. When two or more matches occur from a place or transition, the activation will branch. Coloured Petri nets can therefore be viewed as some form of multi-state machine, making them highly suitable for MAS and concurrent, distributed processes in general.

[1] The use of different colours for example is already incorporated into OutOfBrain. It is one of the features that improves usability greatly.

3.1.5 Coloured Petri nets in MAS

[Weyns and Holvoet 2004] is a study that utilizes the power of coloured Petri nets in a MAS. They construct a coloured Petri net for a simple domain with two agents known as the packet world. The domain was first introduced by Huhns and Stephens in 1999 as a research topic for investigating sociality in MAS. From [Weyns and Holvoet 2004] it is easy to see that coloured Petri nets would also do a good job modelling much more complex systems with any number of agents, due to their modularity.

The packet world consists of a two-dimensional grid with two types of packets distributed randomly on it. Furthermore, there are two identical agents with a limited view of the world and two goal-squares, one for each type of object. The agents have to pick up the objects and deliver them to the appropriate goal one by one. One set of experiments is conducted with communication and another set without communication[1]. The results are compared to find out whether communication increases performance. The measure of performance is a simple counter that increments whenever an agent makes a step, picks up a packet or puts one down. The task is done when all objects have been delivered to their goal-squares. Actions are not always successful; two agents can try to pick up the same packet simultaneously or try to step to the same square. In such cases, only one of the agents will succeed. Uncertainty over the outcomes of actions is one of the reasons why the packet world is a good research topic for studying MAS. A real world example of uncertainty over action outcomes is the unreliability of communication channels.

Figure 3.2 provides an object-level view of the entire system presented in [Weyns and Holvoet 2004]. It consists of five separate coloured Petri nets that can interchange data through interface places (ovals). These work much like the interfaces of objects in object oriented programming. An interface place is simply a data storage that copies its content to all interface places with the same name in other Petri nets. For sake of simplicity only one interface place is drawn per data interchange channel. The five nets represent the environment, agent 1, agent 2, the synchronization module and the postal service. How these nets work internally is beyond the scope of this thesis, for the full version of the Petri net for the packet world we refer to [Weyns and Holvoet 2004]. We can say however, that their internal mechanism resembles object oriented programming with methods inside objects dividing the work and passing parameters to each other. The power of coloured Petri nets becomes apparent when used in parallel computing. They can be viewed as multi-state machines. Each net can be in several states (places) simultaneously and a system consists of several nets that can all be active at the same time.

The interaction between the five different nets goes somewhat like this. First, both agents have to perceive the world. They do this by reading out the data at interface places i4 and i5 respectively. The internal mechanisms in each agent deduce which action they should perform, given the observation of the environment. For example, if a packet is in view, the agent should step toward it if it is not yet holding one. Before the agents choose an action, they communicate to gain information about the invisible part of the environment and to make sure they will not get in each other's way. Via interface places i2 and i3 the agents can send and receive messages to and fro. All messages pass through the postal service to ensure that everything goes according to plan. i.e. messages could get corrupted on the communication channel and an agent should not be able to send a message when the postal service is busy passing a message toward him.

[1] In the experiments without communication, the agents communicate implicitly by making changes to the world that the other agent can perceive. Where exactly the border lies between implicit and explicit communication is not clear. It is a source of much debate in the field of robotics.

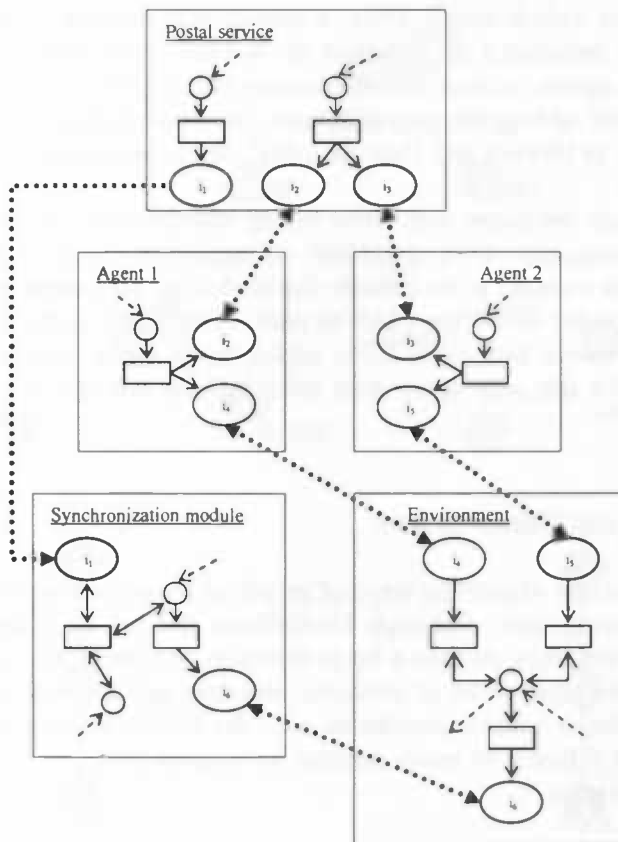


Figure 3.2: Object-level representation of the packet world. We focus on the interfaces between the Petri nets, leaving out representations of the internal mechanics. Ovals are interface places, taking care of data transfers between the five different nets. Circles are internal places and rectangles are transitions. We refrain from depicting actual data in this figure for sake of simplicity.

After observation, communication and reasoning, an action is chosen by each agent. The environment reads out data about which actions the agents try to perform through interface places i4 and i5. It checks whether the actions are successful and sends back the results for the agents to consume. The synchronisation module ensures via i1 and i6 that all activities are properly closed off before another cycle of observation, communication, reasoning, action and result consumption is initiated.

The results Weyns and Holvoet present prove that communication does indeed increase performance in the packet world in terms of energy consumption by the agents. They also feel that the results can be generalized to more complex tasks. This feeling is supported by experimental results from other studies, e.g. [Stone and Veloso 2000]. It indicates that agents in a community need communication in order to perform well. A significant increase in performance only occurs when the agents can perceive just a small part of the environment. This makes sense, because situations in which agents cannot choose a good action from their own observations occur more often when visibility is bad. For example, agent 1 could be holding a packet already. He could then tell agent 2 (provided they can see each other) that a packet is present to the east, beyond the visible reach of agent 2.

Results from experiments with and without communication are compared. This approach can also be found in [Stone and Veloso 2000]. Here, a similar experimental setup is used; the predator-prey domain, first introduced by Benda et al. in 1986. Stone and Veloso start out with a set of homogeneous agents without communication skills. Then they make the agents ever more sophisticated, first adding heterogeneity and communication separately and then adding them both. Just like in [Weyns and Holvoet 2004], the more sophisticated the agents are, the better they perform.
Weyns and Holvoet conclude the paper with some strong statements. First of all, they state that coloured Petri nets have expressive graphical representations with good structuring facilities. Secondly, they are claimed to be suitable for modelling concurrent processes with communication channels and for modelling MAS as well. Thirdly, the authors point out that their Design/CPN tool can prove that every finite packet-world has a correct solution in a limited number of steps. We can only agree with these conclusions, given the clarity and validity of their paper.

3.1.6 Introduction to the OutOfBrain system

Now that we have a clear picture of existing ways of graphical programming, we can turn our attention to the OutOfBrain system. Although OutOfBrain did not originate from graph rewriting or coloured Petri nets, they do have a lot in common. Moreover, the way knowledge is represented in OutOfBrain reminds us of semantic networks and the way an OutOfBrain programmer has to think shows some resemblance with the PROLOG way of thinking. In other words, OutOfBrain is a fusion of many popular techniques from the field of artificial intelligence and computer science.

Figure 3.3 shows the OutOfBrain equivalent of the rewrite rule application in Figure 3.1. The top part is the rewrite rule and the bottom parts are the host graph and the result graph. The node labelled <process> indicates that the rewrite rule is a separate process that operates in parallel with other processes. The one labelled <implication> tells the OutOfBrain interpreter that we are dealing with a rewrite rule. It has both a <premise> edge and a <conclusion> edge, pointing to the subgraph to search for and the subgraph to replace it with respectively. The <group> labels are necessary to indicate which nodes are part of these subgraphs.
Finally, the <transition> edges contain embedding information. They indicate which nodes in the result graph correspond to which ones in the host graph, so that the replaced parts can be glued in place appropriately. Figure 3.1 describes a mechanism that uses geographical gluing. It derives which nodes in the result graph correspond to which in the host graph from their geographical location. OutOfBrain uses explicit gluing, thus it does not need to know the geographical locations of the nodes. With explicit gluing, subgraphs match easier, because their geography does not need to be identical. This can work to the programmer's advantage, but it can also lead to unexpected matches. Without the use of labels, the rewrite rule in Figure 3.3 would have found three matching subgraphs instead of just one. The OutOfBrain manual can be found in Appendix A. It contains descriptions of all predefined labels, as well as documentation on how to use extra features like preferences, normalities and arithmetic.

Figure 3.3: OutOfBrain version of Figure 3.1. On screen, different types of nodes and edges are depicted in different colours to improve usability. Above is the actual rewrite rule, below are the host graph (left) and result graph (right). Note the use of labels to make sure that the rewrite rule does not match any other subgraphs.

One might wonder why OutOfBrain uses a graphical tool for implementation while most other programming languages use textual representations. Indeed, for object-oriented programming, text is much more convenient a medium than graphical representations. However, OutOfBrain is not an object-oriented programming language. The choice of the graph as universal data-structure[1] practically implied the use of a graphical implementation tool. It provides the programmer with a good overview of the entire graph. Textual representations of graphs would not.

[1] OutOfBrain's universal data-structure for artificial intelligence applications is based on a context-dependant hierarchy. Unlike semantic networks, OutOfBrain does not determine the hierarchical properties of an object, until the context is given.

The current version of OutOfBrain offers only central control. All rewrite rules and host graphs are represented in a single file. A version that can handle geographically distributed control is currently under development. Developers, human agents and software agents will soon be able to work together using OutOfBrain, even when they are located on different continents. For the current project, we simulate distributedness by posing restrictions on what data the agents can access. The agents have to communicate in order to receive information from one another. After this short introduction to the OutOfBrain system, it is time to compare it to the graphical programming methods described in Sections 3.1.1 through 3.1.5.

3.1.7 Relating OutOfBrain to other graph rewriting systems and Petri nets

Now that we have some knowledge of what goes on in the field of graphical programming, we can try to give OutOfBrain a place in this larger picture. Although OutOfBrain did not originate from existing graph rewriting techniques or Petri nets, we argue that it has a lot in common with both.

Comparison to graph rewriting
In Figure 3.1 we presented a graphical representation of general graph rewriting according to [Blostein et al. 1995]. All commonly accepted graph rewriting techniques fit this description. In Figure 3.3 we showed that this is also the case for OutOfBrain. This is good news, because it allows us to draw upon results from the field of graph rewriting. Of course, we cannot simply say that everything that holds for graph rewriting also holds for OutOfBrain. As stated before, the programming language that is being developed at Rockingstone did not originate from graph rewriting. Furthermore, there are many forms of graph rewriting. All with different properties. Thus, we have to be careful when drawing analogies.

In OutOfBrain, gluing information has to be stated explicitly in the form of 'transition' edges between nodes in the g_l and the g_r of a rewrite rule (see Appendix A, 5.6). When comparing OutOfBrain to other graph rewriting mechanisms, it turns out that OutOfBrain is relatively unconstrained. Normally, this would lead to poor formal properties (Section 3.1.2). General subgraph isomorphism (searching for applicable rewrite rules) is a well known NP-complete problem [Andries et al. 1999]. However, since the search engine uses a database that includes a unique index for each node, search times are decreased considerably. Note that worst-case time-complexity is still intractable. It is the average time complexity that becomes workable thanks to the database approach. Space-complexity suffers from the database approach, but since time-complexity is the bottle-neck in most cases, this is acceptable. In combination with a finite domain, the use of unique indices leads to tractable solutions.

OutOfBrain utilizes another clever hack to increase its speed. Each node in a rewrite rule has a number of candidate nodes in the host graph. The simplest way to check a rewrite rule's applicability is to check all permutations of the set of nodes in the rule's precondition for applicability in random order. However, this approach definitely yields an exponential worst-case time-complexity ($O(c^n)$ where n is the total number of nodes in the host graph). Hence, it is better to use some search heuristic instead of random applicability checks. The idea is to check the node with the least candidates first. In most settings, nodes that have labels and many connections with other nodes are apt to have less candidate nodes in the host graph. This is because they are more specific than unlabelled nodes (wildcards) with less or no edges. This idea can shrink search-trees considerably. However, just like with the idea of uniquely indexed databases, worst-case time-complexity remains the same. It is the average time-complexity that improves. The current version of OutOfBrain uses depth-first search for

subgraph isomorphism[1], but future versions will utilize a more sophisticated algorithm that combines depth-first with breadth-first search. See [Russell and Norvig 1995] for an overview of existing search algorithms.

Comparison to Petri nets
When examining the literature on Petri nets, we soon found that it does not have a lot in common with graph rewriting. Both are graphical programming languages, but that is where the similarity ends. However, we did find studying Petri nets useful for the following reason. Coloured Petri nets are a form of parallel processes. Since rewrite rules in OutOfBrain are organized in parallel processes as well, the global structures of the two systems show great resemblance. In graph rewriting, rewrite rules are not necessarily organized in parallel processes, so on a global level of description, OutOfBrain is closer to Petri nets than to graph rewriting.

3.2 Communication between agents

One of the goals of this project is to develop a better understanding of sociality in MAS. We agree with [Stone and Veloso 2000] that social skills are necessary in any MAS application that is at least moderately complex. It is due to this goal that we choose an approach where a team of agents has to communicate in order to work together, although such an approach is not the best choice from an engineering point of view. As stated before, the agents are not geographically distributed. Instead, we simulate distributedness by posing constraints on the agents' accessible data. After all, because of our research goals, we want to create a system that is as close to true MAS as possible.

The Foundation for Intelligent Physical Agents (FIPA)[2] is a committee that develops standards for MAS. FIPA was founded in 1996 and memberships include several academic institutions as well as a number of companies. Due to lack of commercial support, it was transformed to an IEEE standards committee in 2005. The only standard developed by FIPA that was widely adopted by the research and business community is their Agent Communication Language (FIPA-ACL) [FIPA 2001].

This formal language relies on *speech act theory* which was developed by John Searle in the sixties [Searle 1969]. His theory was inspired by the work of John Austin and it was enhanced in the 70s by Terry Winograd and Fernando Flores. The main idea behind speech acts is that utterances of language have intentions associated with them. They can do more than just making factual assertions. When I ask someone a question for example, I intend to receive the correct answer. This is why utterances can be seen as a subclass of motor actions instead of just a tool for asserting facts.

The components of a message in FIPA-ACL can be divided into three parts: performative, content and housekeeping [Wooldridge 2002]. The performative contains information about the intention of the sender (e.g. request, inform), the content holds the information that is actually grounded in the environment and housekeeping information ensures a correct transfer and interpretation (i.e. sender, receiver, language and ontology).

The content and performative together determine what an agent is trying to say. The content "the door is closed" for example will be interpreted differently when the performative

---

[1] Depth-first search is possible because the depth of the search-tree is known and finite. It is determined by the size of the subgraph that needs to be checked for applicability.

[2] The name FIPA suggests that it is a foundation that aids the development of physical robots. However, FIPA has never devoted much attention to the physical aspects of agents.

is changed. If the content is accompanied by the performative "inform", the message becomes "The door is closed." But if the performative is "request", it will be interpreted as "Could you close the door?" There are many more predefined performatives and designers can also introduce new ones, although this is not encouraged by the FIPA committee. The sender and receiver attributes make sure that message transfers go according to plan. Finally, the language and ontology of a message ensure an appropriate interpretation of the content and a common frame of reference (symbol grounding) respectively.

3.3 User modelling

The general consensus about learning in the field of artificial intelligence is that it is absolutely necessary as soon as domains become moderately complex [Stone and Veloso 2000]. In most domains it is simply impossible to hard-code everything at design-time. An intelligent computer program has to be able to learn from experience during run-time. Instead of thoroughly studying the literature on user modelling, we draw upon our own knowledge of learning mechanisms.

As stated before, learning in QDT+ amounts to the addition, deletion and augmentation of facts, undefeasibles, preferences and normalities. The latter two in turn lead to reordering possible world clusters as well as the injection of new information into them. The adaptation skills of the diary assistant can use several sources of information. First of all, there is the dialogue with the user. New facts, undefeasibles, preferences and normalities can be derived from the dialogue in numerous ways. Also, old ones can be removed or augmented.

An important tool for this type of learning is the punishment or reward from the user. When new rules about the user and the environment have been discovered, the diary assistant can ask the user if the new information is correct (explicit learning). Furthermore, whenever the team of agents has come up with an action, they could first ask for the user's confirmation. If he or she is not satisfied with the proposed action, the underlying argument should be punished (explicitly), leading to the deletion or augmentation of rules. If on the other hand the user agrees with the proposed action, the argument should be rewarded. In this mode of learning, we have to deal with the credit assignment problem. The dialogue with the user can help the system to find out which part of an argument is truly responsible for the user's judgement.

An altogether different mode of learning takes place through the analysis of the user's behaviour. Again, possible results are the creation, deletion or augmentation of facts, undefeasibles, preferences and normalities. The longer our diary assistant analyses the user's behaviour, the better it can estimate what the planning of a typical day or week should look like. This is a comfortable mode of learning, because it is implicit. The tuning of parameters such as the user's energy per day or the desired number of items on the todo-list is also done by statistical analysis. A source of information that is used for both statistical analysis and qualitative learning techniques is the master-apprentice metaphor. When the user is controlling the diary and todo-list manually, the user-agent can watch his or her every action and learns.

The histories of the OutOfBrain graphs can be saved. These histories can be monitored constantly. When a sequence of subgraphs has occurred several times, an agent can construct a rule that captures the sequence and offer the new rule to the user in a convenient form, so that he or she can confirm or reject it. This mode of learning is inspired by ACT-R's production compilation [Taatgen and Lee 2003]. ACT-R is well suited for integration into QDT, because both are inspired by cognition and because they are both largely symbolic.

The last mode of learning we would like to put forward here is the derivation of absent item-attributes. In some cases, the system can fill in certain information on its own. The amount of information per item the user needs to specify should decrease during usage thanks to user modelling. The diary assistant could learn that certain activities are fun or important and others are not. Hard-coded derivations are also possible: e.g. urgency should increase when a deadline is close and movability of an item should decrease if there are other people involved. This concludes the introduction of possible learning modes. The final Section of the current Chapter discusses the control and display components of our adaptive diary assistant. While designing the interface, we again draw mainly upon our own knowledge.

3.4 Interface design

As computers take on an ever increasing role in our daily lives, the need for user-friendly interfaces is greater than ever. Human-computer interaction is currently one of the fastest growing disciplines, both in science and in business. One of the most interesting examples thereof is the ambient intelligence movement described in [Aarts and Marzano 2003]. According to the ambient intelligence philosophy the era of houses infested with grey and black boxes is over. Devices should be integrated in the environment and the user should play a more central role. Aarts and Marzano also plead for simple but intelligent interfaces that change to fit the user's needs. This plea is backed by many other researchers (e.g. [Rich 1999]). Displays should not contain inappropriate information and controls that are not used should be hidden. We do not have the intention of creating a perfect interface. Instead, we try to utilize the knowledge on interface design that is already at our disposal.

3.4.1 Control components

We intend to use a conversational text-interface to control the diary assistant. We are aware of the fact that a proper conversation text-interface like the ones used in modern chatbots requires extensive natural language processing. Since this is not one of our research topics, we choose to only allow a language with a formal syntax instead of natural language. Furthermore, we support text-input with other forms of control, because this makes for a more convenient interface. If only text were allowed, users would have to memorize a lot of commands. These other forms of control could be anything from radio-buttons to slides. In our design, we rely on our own factual knowledge. Rockingstone has a lot of experience with developing database applications, which is also a welcome source of knowledge for interface design.

3.4.2 Display components

Just like the controls, the display components should be as simple as possible while still being able to display all the necessary information. To achieve this, some information needs to be hidden. This can be done by using drop-boxes. Furthermore, the contents of the display should be user-dependent, but it should not be infinitely dynamic, because users also look for a certain stable factor in the interface [Rich 1999]. Available options and user-specific domain information are typical display components that can be determined dynamically. The position of buttons on the other hand, should not change. This concludes our brief discussion of the interface for now. We pick it up again in Section 4.7.

4 Design

From the literature covered in Chapter 2, we construct the combined architecture QDT+ that is as general purpose as possible while being fit for the particular application we have in mind: an adaptive diary assistant. We want to be able to reason about time, preference and normality in one and the same framework. QDT provides linear preference and normality orderings, while BDI gives us time structures that branch into the future and are linear in the past.

In Section 4.1, we briefly introduce the QDT+ model and logic. We also show that we need the advantages of qualitative decision theory described in Section 2.2. The adaptive diary assistant is composed of seven parts; three agents and four parts of the environment. Section 4.2 explains these seven parts and their interrelations. A worked example of how to determine goal worlds from preference and normality rules is presented in Section 4.3. In Section 4.4, we finally arrive at describing the diary assistant's methods and data. The techniques we use for communication and learning are explained in Sections 4.5 and 4.6 respectively. Finally, Section 4.7 presents the user interface of the diary assistant.

4.1 Introduction to QDT+

In an adaptive diary assistant, agents have to be able to reason about time. In particular, they need to make statements about the dates and times in the diary. As already explained in Section 2.5, we do not use modal logic for this. Instead, we choose to represent dates and times using first-order predicates in combination with arithmetic. This is a great tool to define complex statements about dates and times like "later today", "somewhere next week" or "every weekend". However, we do use a modality for time in a different matter. The histories of the environment and the agents are stored so that the agents can learn from the past and predict the future. The way we handle reasoning about the past and future of the system (as opposed to the dates and times in the diary) is practically identical to the way it is done in the BDI architecture.

The agents also need to formulate their preference and normality orderings toward possible worlds for the following reasons. Defeasibility is necessary, because new knowledge comes in constantly through input from the user and communication between the different agents. There is a reasonable amount of uncertainty, especially when trying to model the user in a QDT+ agent. Complexity considerations are also important, since we have three modalities for each agent and an environment that is reasonably complex. Quantitative methods would not do well in such a complex environment. In fact, they would probably yield intractable solutions: worst case time complexity and average time complexity would be too high.

We compare the diary environment to the real world using the classification dimensions from [Russell and Norvig 1995]. Complexity is not as great as in the real world, because the diary environment is discrete and relatively simple. Nonetheless, it is a dynamic world, the different agents do not have full access to it, and non-determinism can probably not be fully eliminated. These are more than enough reasons to choose qualitative methods and they are also good motivations for choosing an architecture like QDT, BDI or a combination thereof that has a high level of expressive power. To summarize, we end up with three modalities: preference, normality and time. These modalities can be combined in complex formulas and they operate upon normal predicates and arithmetic-predicates representing events, dates and times.

4.2 Applying the QDT+ model to a diary assistant

In Section 2.5 we have shown that QDT [Boutilier 1994] and BDI [Rao and Georgeff 1991] can be combined without problems. Mehdi Dastani and his colleagues have studied both architectures extensively and they have come to the conclusion that QDT and BDI are likely to be fully compatible [Dastani et al. 2003]. Designers can take the parts they need from BDI and QDT and combine them into a new architecture and possible worlds model that fits the task at hand. In this section we will present a QDT model and logic extended with the time component from BDI. This approach of adding parts of BDI to QDT is exactly the suggestion Dastani and his colleagues made in their 2003 paper. (They also state that taking BDI as a starting point and extending it with QDT components is another possibility.)

However intuitive the distinction between desires and intentions provided by [Rao and Georgeff 1991], we do not need it. Boutilier's preference orderings serve much the same function and they provide the possibility for many levels of desirability instead of just two (desire and intention). In Boutilier's approach, the cluster of worlds with the highest desirability (the one that is at the bottom of the preference ordering) becomes an intention.

The concept of obligations [Broersen et al. 2002] seems a useful addition to our model, because the agents reason about appointments and todo-items. However, we have chosen not to incorporate this concept for two reasons. First, we want to come up with a design that obeys Occam's razor: the model should be as simple as possible. Second, we feel that obligations are so fundamental in a diary environment, that it would be best to consider every proposition on the diary as an obligation, thus rendering the introduction of a new concept obsolete.

4.2.1 The agents and their environment

A QDT+ model $M = <W, V>$ consists of a set of possible words W and a valuation function V that assigns truth values to predicates in possible worlds. The elements of W are of the form $w = \{d, t, dch, tch\}$ where d stands for diary, t stands for todo-list and dch and tch are the diary conversation history and todo conversation history respectively. When the human user communicates with the adaptive diary assistant, messages are being passed from user-agent to diary-agent and back. These messages are the content of the diary conversation history. Communication with the todo-agent is stored in the todo conversation history.

Our set of agents operating on this model is $A = \{da, ta, ua\}$. Agent da is the diary-agent, ta is the todo-agent and ua is the user-agent. Only the user-agent has a dynamic knowledge base. However, the other two agents can still derive new information about the user and the world around him or her. In fact, they can discover things that the user-agent could never discover, because they have access to different parts of the environment and their

own, unique inner mechanisms. The only difference is that new information found by the diary-agent or todo-agent is sent to the user-agent, so that he can put it in his knowledge base. When learning comes into the picture, the agents need the modality time (*T*) to express events in the past and the future. The complete set of orderings of all agents is therefore $O = \{ \leq_P^{da}, \leq_N^{da}, \leq_T^{da}, \leq_P^{ta}, \leq_N^{ta}, \leq_T^{ta} \leq_P^{ua}, \leq_N^{ua}, \leq_T^{ua} \}$.

For the possible worlds semantics of these orderings, we refer back to Sections 2.3 and 2.4. It is important to realize that the dates and times in the diary itself are not represented in *T*. Instead they are formalized in arithmetic-predicates. Figure 4.1 shows how the agents can interact with each other and with the environment (possible worlds). All three agents have the ability to communicate with the other two agents. Among other things, they can query each other for information and they can send requests to execute a certain action. The most important aspect of a multiagent team is that the members have different capabilities. In particular they have access and control over different parts of the environment. The user-agent models the human user, therefore he should have access and control over more or less the same parts of the environment as the human user[1].

His scope of information-access covers the current view of the diary window and todo window as well as the most recent entries in both dialogue histories (say five or six). Along with the history of his own inner mechanisms, these are exactly the parts of the environment of which the histories are stored by the user-agent and that can consequently be used by him for learning. This means that the user-agent does not have access to the entire diary and todo-list. There is simply too much information for a human being to remember[2]. Instead, the user-agent (who mimics the human user) relies on the diary-agent and todo-agent to provide him with information.

The user-agent has limited control over the system as well. He can only add an entry to one of the conversation histories. The user can also control the diary and todo-list directly, without taking the team of agents into the loop, but then, the system is controlled by the human user directly. The user-agent remains passive. The diary-agent has access to the diary and the entire diary conversation history. He can have control over the environment by adding an entry to the diary conversation history and by adding an entry to the diary itself. The todo-agent operates much the same way, having access to the todo-list and the todo conversation history. He has control over the todo-list and its conversation history.

4.2.2 Reasoning about the past

The agents and the environment are actually complete histories (*T*). The agents need these histories for their learning mechanisms. We are aiming at a user-agent that mimics the human user, so one might say that the histories he has access to should not remain in memory forever. This is a source of debate however. The human user cannot produce the entire history on demand, but when he or she is presented with a particular situation, the user will probably remember that it has occurred before. In humans, this priming effect is quite strong; people remember more than they know. The implicit memories just have to be triggered somehow

---

[1] This resemblance between user-agent and user is hard to achieve since the human user can perform active perception. The search for information can be guided by controlling the interface (e.g. changing days in the diary). For the user-agent to be a good reflection of the human user, we need to design methods for active perception as well.

[2] There may be hundreds of items on the diary and the todo-list. Each item has a number of attributes (e.g. deadline, amount of fun and location). It should be obvious that a normal person can never remember all these items and their attributes. After all, that is why we use diaries and todo-lists.

[Roediger 1990]. Making the entire conversation history available to the user-agent's learning mechanisms is therefore justified.

However, as mentioned above, only the last couple of entries in the conversation are available to the user-agent during normal reasoning. The histories of the diary and todo-list are harder to remember, even with priming effects, because of the nature of the information in there. It is concerned with times, dates and attribute settings that a normal user and therefore the user-agent cannot remember. To retain biological plausibility, they have to rely on the diary-agent and todo-agent to supply this kind of information or facts and rules derived from it.

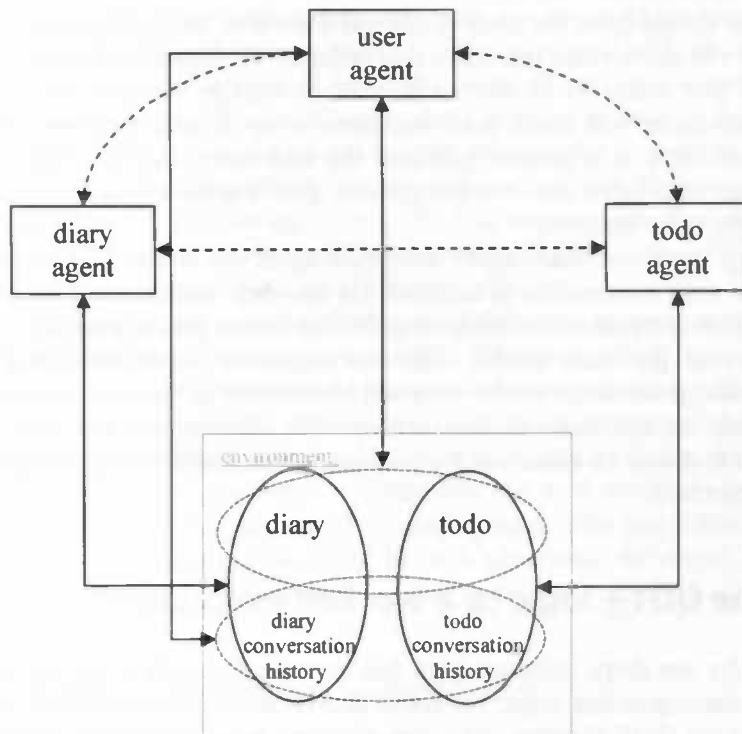


Figure 4.1: Overview of the information flows between the seven parts of the system. Dashed arrows indicate communication channels, solid lines stand for perception and action. The ovals that represent the data that the user-agent can perceive and act upon are dashed, because the human user can only see the information in the diary that is currently on display. Furthermore, he or she can only remember the last parts of the conversations. However, during learning, the user does have access to the entire conversation histories, due to priming effects. Also note that all three agents can perform actions on the conversation histories only by adding utterances to the end of them.

In the time series, each time step is an entire graph representing an agent or a part of the environment. The agents use this information about the past to make predictions about the future and to learn which inferences are valid for this particular user and which are not. To some extent, we also have to keep track of time branches that were not followed, because the agents can learn from those as well. A large amount of data is stored while maintaining these histories, therefore we have to use an efficient storing mechanism that only keeps track of the differences between two consecutive graphs. For OutOfBrain's search algorithm this is no problem, because from outside the database it looks as if the graphs are stored in their entirety.

4.2.3 Cooperation strategies

It is time to make explicit the cooperation strategies used by the three agents. The user-agent mimics the user. There is one conversational text-interface through which both the diary-agent and the todo-agent communicate with the user-agent (and the human user). When the user wants to manipulate the diary, the user-agent starts communicating with the diary-agent, because he wants him to do something for him (e.g. add an entry to the diary). Similarly, when the user wants to control the todo-list, the user-agent will initiate a conversation with the todo-agent. Note that the user can also manipulate the diary and todo-list manually. In this case, the team of agents is not involved.

The user-agent should have the same goals and beliefs as the human user. Over time, their mental attitudes will show more and more resemblance. Consider for example, the times of the week that the user wants to be free or how much energy the user has on a typical Wednesday. The diary-agent will make sure that there is no overlap between diary-items. Furthermore, he should have as a general goal that the user-agent and thus the human user achieves his goals. The todo-agent also has this general goal. Furthermore, he has the goal to minimize the size of the todo-list.

The knowledge bases of diary-agent and todo-agent are static. However, they do provide the user-agent with information to facilitate his learning mechanisms. When the user-agent needs to view a certain part of the diary or todo-list, he can just request the appropriate information from one of the other agents. The diary-agent and the todo-agent can also communicate their findings about the preferences and normalities of the user derived from the changes in diary, todo-list and conversation history. The diary-agent and todo-agent are autonomous, however it should be clear that the user-agent has a higher level of authority and autonomy than the other two.

4.3 Applying the QDT+ logic to a worked example

Now that the model for our diary assistant team has been made explicit, we can turn to the concretization of the corresponding logic. As stated before, the logic we use for our team of agents is based firmly on QDT (Section 2.3). The qualitative decision architecture BDI has been criticized as not having much practical relevance [Rao and Georgeff 1995]. However, this critique has been countered by several successful implementations of the BDI architecture: PRS, dMARS, JACK, JAM, Jadex, AgentSpeak(L), 3APL, Dribble and Coo-BDI. In our study of the literature, we have not found such implementations for the QDT architecture. There have been some attempts to implement QDT, but BDI is notably more popular for practical use. The reason for this is probably that BDI instead of QDT is considered to be the state of the art in qualitative decision theory. QDT has different expressive powers than BDI. First of all, QDT can distinguish between more than two levels of desirability. Whether this is a good thing depends on the situation at hand. Second, QDT uses a non-monotonic logic which is surely an advantage in most domains. Finally, it lacks the modality time which is of course a drawback of QDT.

We agree that BDI has higher biological plausibility than QDT, but from an engineering point of view, we argue that QDT might be a more convenient theory, especially when it is complemented with BDI's way of expressing time. It should therefore not be abandoned just yet. Our design of QDT+ could show that QDT is in fact quite suitable for practical use, just like BDI.

4.3.1 Example: the umbrella problem

We have tested the practical usability of QDT by analysing some toy-problems. The examples in [Boutilier 1994] provided us with a clear picture of how the architecture works. However, Boutilier's work does not specify how exactly preference and normality information should be combined into the determination of goal worlds. The easiest way to do this is by means of a qualitative version of the maximization of expected utility (see Section 2.1). We claim that the method described below yields the same results in many cases as Boutilier's apparatus for calculating action set dominance (Definition 2.18).

Consider the situation (based on Boutilier's example) where I have just heard the weather forecast. The weatherman said that the sky will probably be overcast: $\top \Rightarrow O$. However, I have missed the part where he says whether it will rain or not. Now, let us assume that from previous experience, I know that when the sky is overcast, it will probably rain: $O \Rightarrow R$. After consulting this knowledge, along with the preference for taking an umbrella when it is raining, $I(U|R)$, I should decide to grab an umbrella before I go outside. I also know that when the sky is clear, it probably will not rain: $\neg O \Rightarrow \neg R$. Furthermore, I prefer not to take an umbrella when it is not raining: $I(\neg U|\neg R)$. The decision tree that corresponds to this situation is depicted in Figure 4.2 with solid lines representing my actions (U and $\neg U$) and dashed lines representing uninfluencables (O, $\neg O$, R and $\neg R$).

It is important to realise that normalities are concerned with the uninfluencables and preferences are about the agent's actions. [Boutilier 1994] does not state explicitly how normalities and preferences should be combined into a measure of utility. However, he does say that an agent should first determine all default consequences of his knowledge base (using normality rules) and then reason with this information as if it were true, along with available preference information in order to determine the goal world. In most complex situations, a path in a tree of events consists of interchanging actions and influencables. To show how an agent could determine a goal world in such situations, we return to our example described above.

4.3.2 Possible worlds and their expected utility

There are $2^3 = 8$ possible worlds in this simple example. These possible worlds can be ordered according to their likelihood and preferability. Table 4.1 contains all information about which normality rules are satisfied or violated by the possible worlds. In the rightmost column, the normality ranks of the worlds are indicated. Table 4.2 does the same for preference rules. When we combine the normality rank and preference rank into a measure of expected utility by multiplying their normality and preference rank, we immediately see that world w_1 is our goal world. It has an expected utility rank of $util_rank(w_1) = norm_rank(w_1) \cdot pref_rank(w_1) = 1 \cdot 1 = 1$.

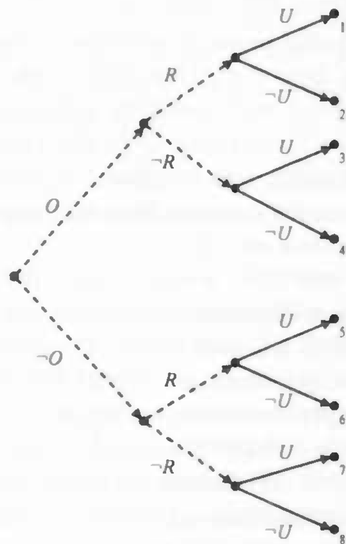


Figure 4.2: Decision tree for the overcast-rain-umbrella example. Dashed lines are uninfluencables, solid lines are actions[1]. The utility of the eight different worlds can be determined using the available normalities and preferences.

world #	$\top \Rightarrow O$	$O \Rightarrow R$	$\neg O \Rightarrow \neg R$	normality rank
w_1	√	√		1
w_2	√	√		1
w_3	√	χ		2
w_4	√	χ		2
w_5	χ		χ	3
w_6	χ		χ	3
w_7	χ		√	2
w_8	χ		√	2

Table 4.1: Calculation of each world's normality rank from the satisfaction, neutrality or violation of the available normality rules.

world #	$I(U\|R)$	$I(\neg U\|\neg R)$	preference rank
w_1	√		1
w_2	χ		2
w_3		χ	2
w_4		√	1
w_5	√		1
w_6	χ		2
w_7		χ	2
w_8		√	1

Table 4.2: Calculation of each world's preference rank from the satisfaction, neutrality or violation of the available normality rules.

[1] The tree suggests that the agent chooses whether or not to take an umbrella after the uninfluencables overcast and rain have happened. However, the agent does not have knowledge about the truth or falsity of **R**.

world #	normality rank	preference rank	expected utility rank
w_1	1	1	1
w_2	1	2	2
w_3	2	2	4
w_4	2	1	2
w_5	3	1	3
w_6	3	2	5(6)[1]
w_7	2	2	4
w_8	2	1	2

Table 4.3: Calculation of each world's expected utility rank from its normality rank and preference rank.

4.3.3 Summation over possible outcomes

When we examine the classical equation for the expected utility of an action (Section 2.1), we see that it sums over the possible outcomes of an action to handle uncertainty. In the example above, this is not the case. However, we can augment the problem so that it fits the classical equation. Consider the situation where an agent has to decide whether or not to take an umbrella and he knows that the sky is overcast. However, the agent does not know that it will probably rain when the sky is overcast or that it should stay dry when the sky is sunny. Figure 4.2 can be reduced to only its upper half in this situation.

The set of possible worlds W consists of only $2^2 = 4$ elements: $\{\{R, U\}, \{R, \neg U\}, \{\neg R, U\}, \{\neg R, \neg U\}\}$. The truth of R is not influencable, which corresponds to the fact that the outcomes of the actions $does(U)$ and $does(\neg U)$ are uncertain[2]. The expected utility of these two actions can be calculated as follows. First, we need to recalculate the normality rank and preference rank. Because we have already devoted considerable attention to how ranks are calculated from statements of the form $I(B|A)$ and $A \Rightarrow B$, we will not go through this process again for this example. We simply provide the ranks directly in Table 4.4. After $done(U)$, the agent can be in either w_1 or w_3. After $done(\neg U)$, the agent can be in either w_2 or w_4.

The expected utility of $does(U)$ is:
$util_rank(does(U)) =$
$util_rank(w_1) + util_rank(w_3) =$
$norm_rank(w_1) \cdot pref_rank(w_1) + norm_rank(w_3) \cdot pref_rank(w_3) =$
$1 \cdot 1 + 2 \cdot 1 = 3.$

The expected utility rank of $does(\neg U)$ is:
$util_rank(does(\neg U)) =$
$util_rank(w_2) + util_rank(w_4) =$
$norm_rank(w_2) \cdot pref_rank(w_2) + norm_rank(w_4) \cdot pref_rank(w_4) =$
$1 \cdot 2 + 2 \cdot 1 = 4.$

The agent will decide to take his umbrella with him.

[1] Because we are dealing with ranks, there should never be gaps between them (between 4 and 6 in this case). After the expected utility ranks have been calculated, they gravitate towards 1. That is how ranks work, every integer in the range from 1 to the lowest rank (5 in this example) must be occupied.

[2] We refer to Section 2.4 for the intended interpretation of these formulas.

world #	normality rank	preference rank	expected utility rank
w_1	1	1	1
w_2	1	2	2
w_3	2	1	2
w_4	2	1	2

Table 4.4: Calculation of expected utility in the new situation: The sky is overcast, but there is no knowledge about the consequences thereof. It could either rain or stay dry, the agent has to consider both to calculate utility.

Note that this example greatly resembles the prisoners dilemma. Taking an umbrella or not is equivalent to cooperating or defecting. Whether it is raining or not is analogous to the opponent's choice to cooperate or defect. One can draw the analogy even further, to the iterated prisoners dilemma. Previous outcomes can be used to guide the current choice. For example, if it was raining this morning, it will probably be raining at noon as well. This analogy shows once again that, despite its simplicity, the iterated prisoners dilemma can teach us a great deal about rational agent behaviour.

4.3.4 Rule priority

The power of this qualitative version of expected utility measurement becomes apparent in complex domains, where both normality ranks and preference ranks on possible worlds can have many levels. In such domains, the method comes up with a single goal world in most cases without ever associating rules (of the form $I(B|A)$ and $A \Rightarrow B$) with ranks or real values, thus reducing the complexity and increasing the intuitiveness of the system. Each rule is equally important. This is no problem as long as the agents have enough information about the domain. There should be a complex network of rules influencing each other's applicability.

However, it is easy to imagine domains where some rules are more important than others. The association of rules with ranks or real values could be necessary in such domains in order to decide which actions to perform. We plead for the use of only a few ranks instead of integers or real values to maintain the qualitative nature of QDT+. Recall from Section 2.4.3 that the BOID architecture uses a prioritized default logic. Rules can be of different types and depending on what kind of agent we need, certain types have priority over others. Social agents for example, give priority to obligations over desires. Obligations are therefore selected first and they can easily prevent desires from firing. We can do the same in QDT (statically and/or dynamically). Some preference rules could be labelled as "like to", others as "love to". Normality rules can be divided into two groups in the same way by labelling them as "maybe" and "probably". More than two groups can be used if necessary.

There are three normalities at the agent's disposal (Table 4.2). From our own common sense we know that the rule $\neg O \Rightarrow \neg R$ is more likely than $O \Rightarrow R$. This is not reflected in our example, because the rules do not have ranks or real values associated with them. Likewise, $I(U|R)$ should be more important than $I(\neg U|\neg R)$. Apparently, we could need ranks on rules after all. Therefore, we keep the possibility to do so open in the design of QDT+. We refrain from using BOID's prioritized default logic however, as long as we do not encounter undesirable equilibria in goal world determination. Priorities over rules make for a less qualitative architecture with increased complexity and decreased intuitiveness. Furthermore, when there are many rules influencing each other, an equilibrium could be a true equilibrium and we simply should not distinguish between the fitness levels of the winning worlds.

4.3.5 Some thoughts on qualitativity and complex formulas

Now, this qualitative version of Savage's classical decision theory might seem a little ad hoc at first. Is it not just a quantitative method with integers instead of real values? We argue that it is not. Boutilier's QDT logic ranks possible worlds according to their relative preference and normality. We bind integers to the different clusters of possible worlds in order to combine the orderings into a single utility ordering as described above. The number of levels however, is finite and in most cases quite small compared to the entire range of the integers. The disadvantages of quantitative decision theory discussed in Section 2.2 do not apply, or at least less so, to the method discussed above. The method is of course of a less qualitative nature than BDI, because of the different levels of preferability and normality. This leads to a reduction of non-deterministic decisions. Where a BDI-agent often has to choose between two worlds that are equally good, a QDT-agent can distinguish between the utility of these worlds due to different levels of preferability and normality. Whether or not this is a good thing depends on the situation at hand.

Before we turn to the adaptive diary assistant itself, we need to discuss one more issue. We do not allow for combined preference/normality formulas like $I(B|A) \Rightarrow I(C)$ within a single agent. Instead, we depend on the calculation of utility described above to link preference and normality information. After considering many examples, we came to the conclusion that it is highly unlikely that we need such combined formulas. Besides, combined formulas do not occur in [Boutilier 1994] either. The reason we do not need them is that preferences are always about the consequences of the actions of agents, whereas normalities are concerned with uninfluencables. This strict division is also apparent in Figure 4.2.

Only when agents are reasoning about each other, these combined formulas could occur. An agent can have a preference for another agent to have a certain normality rule for example[1]. The three agents are truly autonomous, they can even have conflicts. For instance, the user-agent modelling a lazy user could have preferences expressing his or her dislike for busy days. However, the static diary-agent always has rules that indicate his preference for doing things as soon as humanly possible. The agents should discuss this disagreement in order to come up with reasonable compromises. This example emphasizes the advantage of a team of agents over a single agent.

4.4 Designing the adaptive diary assistant

For the design and implementation of our diary assistant, we use an incremental strategy. We start with a simple diary assistant that does not possess adaptation skills or advanced planning capabilities. From there, we extend the system until we have something that behaves like a personal planning assistant.

One of our secondary goals is to develop an electronic diary that offers a bit more than existing diary systems. The first extension we propose is the capability of planning. The user can invoke a reasoning process that returns the most urgent todo-item on the list. This process uses the attributes of the todo-items to determine which one should be moved to the diary first. These attributes are: importance, mental effort, physical effort, fun, type, subtype and location. Furthermore, the system can generate a list of possible times and dates for todo-items. And finally, the best suggestion will be chosen from this list using user specific preference and normality rules.

[1] The operators I and $\Rightarrow$ can be accompanied by subscripts indicating which agent's preference or normality rule we are dealing with.

In order to obtain user specific information, we need learning processes. Consequently, the second extension we propose is adaptation. Opportunities for learning are abundant in the planning of one's appointments and obligations. Knowledge of the real world can be used to improve adaptability. For example, the diary assistant could be aware of the geographic locations of the user's office and home, so that it can bear in mind travel times during planning. An exhaustive list of these learning opportunities will be given in Section 4.6.

4.4.1 Preview of the diary assistant's functionality

Before we turn to a detailed model of the entire system, we preview part of its functionality in Figure 4.3 to get a feel for the way things work in OutOfBrain. The part of the system we have chosen for this preview is the planning of todo-items. For sake of simplicity, we do not yet incorporate user modelling into the preview. The planning of todo-items takes place when an item does not have a date and time specified or if the proposed date and time lead to a conflict. The diary-agent has to search the diary for alternative dates and times where the pending todo-item would fit. After this process, reasoning takes place to determine which alternative is the best. This topic will be treated later, along with other planning functionality and adaptation skills.

Four processes explained
An OutOfBrain program is essentially a collection of parallel processes. Like in most implementations of parallel processes, parallelism is simulated on a serial machine. Fairness is guaranteed: all processes will try to fire periodically. In the real application, there are only three processes, one for each agent. The reason for this choice will be treated in Section 6.2. Consequently, the preview presented in Figure 4.3 is not an accurate reflection of the real application. However, it does show how parallel processes are implemented in OutOfBrain. Each node in Figure 4.3 is actually an implication (I) or condition (C) in OutOfBrain.

Process p_1 is activated when it encounters an item in the *request* state. Next, a possible reference edge from a previous item is removed and a new one is created, pointing to the appropriate item. The next condition of p_1 checks whether the day the item should be placed on is empty. If so, the state of the item is set to *assign* and the process ends and becomes available for firing again. If not, the process has to check whether an overlap would occur if the item were to be placed at the desired time. There can be two kinds of overlap: Either the start time or the end time of the pending item (or both) are between start and end of an item on the diary. Whenever one of these overlap checkers returns *true*, the item's state is set to *rejected*. If both return *false*, the item state becomes *assign*. These checks could be done by a single rule using the or-operator. However, OutOfBrain does not have an or-operator, because such an operator does not fit the graph rewriting paradigm. This is of course inconvenient, but OutOfBrain is still functionally complete.

The next two processes are quite simple. Process p_2 is activated whenever it encounters an item in the *assign* state. It then connects the item to the appropriate day. Process p_3 looks for items in the *rejected* state. Whenever it finds one, its state is set to *suggest* and the item's duration is calculated from its start and end attribute for planning purposes.

Finally, p_4 starts running when it finds an item in the *suggest* state. A possible old reference is cleaned up and a new one is created. Then, the process will start searching for

alternative times for the item at nine o'clock in the morning of the desired day[1]. Then, the process enters a loop: it will collect suggestions until it reaches five o'clock in the afternoon or until it has collected ten suggestions. Nine o'clock is compared to the start time of the items already linked to the day. If there is overlap, we jump to the end-time of the overlapping diary-item and continue searching from there. If there is no overlap, a suggestion is added to the todo-item, we jump half an hour, to nine thirty in this case, and continue the search[2]. This concludes our preview of the adaptive diary assistant's functionality.

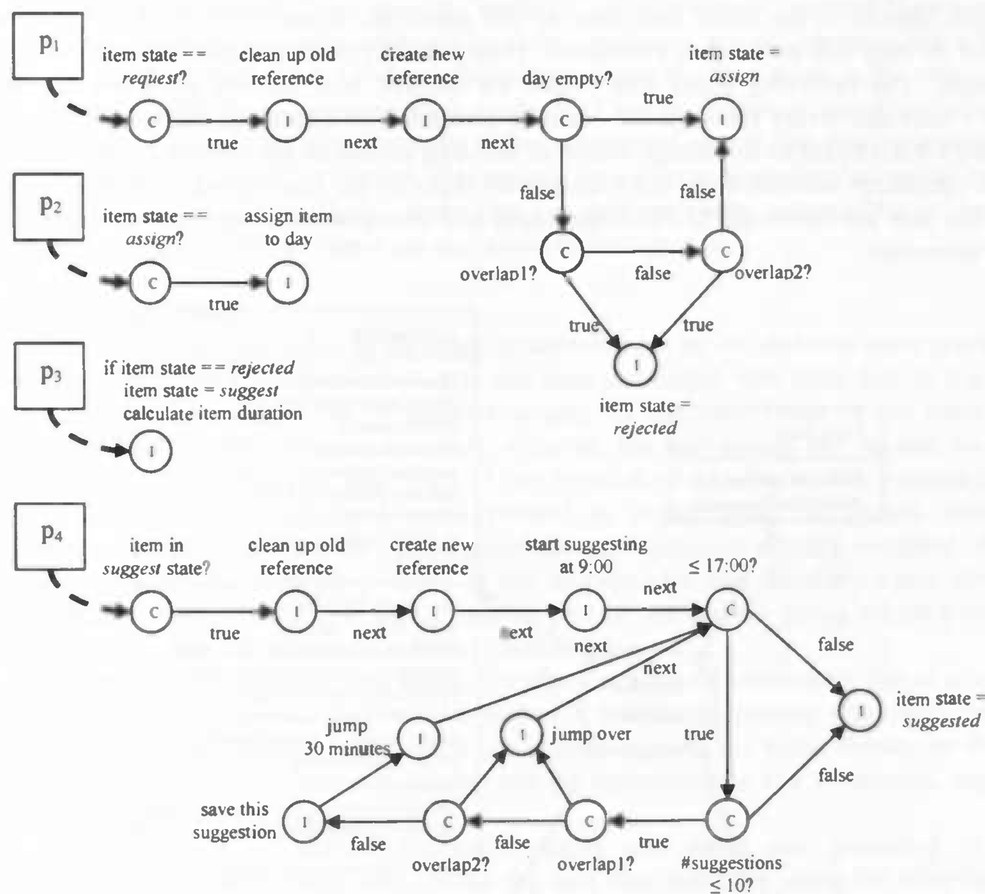


Figure 4.3: Part of the system's basic functionality in four parallel processes. Process p_1 checks whether the pending item fits at the specified date and time. Process p_2 assigns items that fit to the appropriate day. Process p_3 passes rejected items to process p_4, which in turn collects alternative dates and times for the item.

[1] The diary assistant should also be able to make suggestions on other days, but for sake of simplicity, we assume for now that the process only looks for alternatives on the day the item was originally supposed to be added to.

[2] There are situations where this method does not provide good suggestions. Suppose an overlap of five minutes occurs. In this case, we should step back five minutes to yield a good suggestion. In the real application, the agents have more advanced planning skills than in this preview.

4.4.2 Overview of methods

The functionality of the seven components in Figure 4.1 should now be explained. There are three agents and four environment parts. The preview given in Figure 4.3 explains part of the functionality of the todo-agent. Each agent has different tasks and they have access to different parts of the environment. Furthermore, we have an Item class and a Message class. Figure 4.4 shows the three agents along with their methods and accessible data. The data-classes Item and Message are treated in Figure 4.5.

It is important to keep in mind that the three agents are autonomous. They differ from classical objects in the sense that they do not construct objects from each other's classes. Neither do they call each other's methods. They can only communicate their desires through messages. The receiving agent will decide for himself how he will act upon the sending agent's message. Every message has a sender-attribute and a receiver-attribute. Therefore we can add a blackboard to the design where all pending messages are collected. The three agents check constantly whether there is a message for them on the blackboard. Even the commands from the user are converted to FIPA-messages and then posted on the blackboard among the other messages.

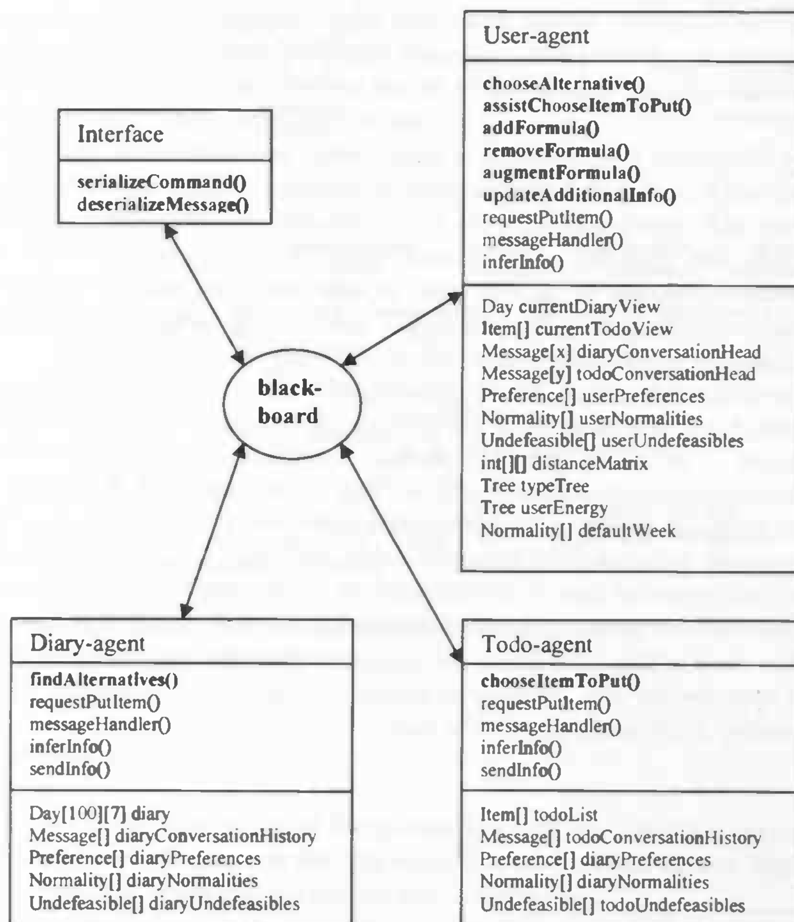


Figure 4.4: The interface, the blackboard and the three agents along with their methods and data-members.

Learning and interface communication
We start our guided tour through the system at the user interface. Commands from the user are converted to messages and then posted on the blackboard by the method *Interface.serializeCommand()*. The receiver of each message is either the diary-agent or the todo-agent, depending on the contents of the command. Whenever information should be presented to the user, *Interface.deserializeMessage()* converts the message containing the information to a command that invokes the display-components of the diary assistant.

Before we explain the processing of the three different types of items, we want to draw the reader's attention to User-agent's methods *addFormula()*, *removeFormula()* and *augmentFormula()*. These methods should be called whenever new information about the user or the world around him or her has been discovered. Since the user-agent is the only one that learns, the other two do not possess such methods. They do however, check periodically if they can find out something new about the user or the world so that this new information can be sent to the user-agent. He will then add, remove or augment formulas that model the user. The method *updateAdditionalInfo()* maintains the data-members *distanceMatrix*, *typeTree*, *userEnergy* and *defaultWeek*. These are explained in Section 4.6.

Processing diary-items
New items that have to be added to the diary or todo-list can be divided into three groups. The first group consists of items that have a date and time associated with them that do not lead to overlap on the diary. These can be added manually by the user. None of the three agents intervenes with this form of manual control. After all, the user should still be able to control his own diary. The method that takes care of this is not shown in Figure 4.4, because it is not part of the multiagent system. Manual control is implemented in Delphi instead of OutOfBrain, just like the interface. If the given date and time are already occupied, the user receives a warning from the diary-agent. If the user decides that the diary-agent should do something about the overlap, the item becomes part of the second group which consists of items with a date and time that do not fit the current diary.

For each item in this second group, the diary-agent will construct a list of alternative dates and times using *Diary-agent.findAlternatives()*. Besides suggesting alternative dates and times for the pending item, the diary-agent can suggest moving the items already on the diary to make room for the new one. Of course, not all appointments can be moved, especially when other people are involved.

The item and its list of alternative dates and times are presented to *User-agent.chooseAlternative()* which will choose the best date and time using the information in the user model. The chosen alternative is then suggested to the human user. If the agents have done their job properly, the user will accept the alternative. But if he or she rejects it, another alternative has to be suggested. Now, from the user's reply and the possible dialogue following it, other alternatives could also be removed from the list. Then, *User-agent.chooseAlternative()* is called again and the best remaining alternative is presented. This process continues until the user agrees and the item can be placed on the diary.

Processing todo-items
The third item group consists of items that do not have their date and/or time specified. The user wants the system to provide an appropriate date and/or time for such items. When such items come in through the interface, they will be placed on the todo-list, along with their specified attributes. Any one of the three agents can decide that it is time to move a todo-item to the diary. The user-agent could decide this, because the human user gave a command to do so. The diary-agent could decide that it is time, because the diary is relatively empty. Finally,

the todo-agent himself can decide to offer an item to the diary, because the todo-list has become too long.

After the todo-agent receives such a request or decides for himself that it is time to move an item to the diary, he will use the method *Todo-agent.chooseItemToPut()* to choose the item that should be offered to the diary-agent. He does this by analysing the item's attributes: fun, importance and deadline. These are explained in detail below. Before actually offering the item, he will consult the user-agent. *User-agent.assistChooseItemToPut()* and the model of the user can help the todo-agent in making a choice that suits the user's preferences. After this joint effort to choose the best todo-item, the diary-agent receives the item. He will use its duration to search for gaps in the diary large enough to fit the item. The alternatives are again selected by *Diary-agent.findAlternatives()* and the best alternative is again *User-agent.chooseAlternative()*.

The super-class Agent

The methods discussed so far make the agents unique. These methods are boldfaced in Figure 4.4. The remaining methods are the same for each agent. Therefore, they are methods of the class *Agent* of which the three agents in Figure 4.4 are sub-classes. The first of these methods is *requestPutItem()*. As mentioned above, any one of the three agents can decide that it is time to move an item from the todo-list to the diary. The second method of the super-class Agent is *messageHandler()*. It analyses collected messages and puts the appropriate mechanisms in motion. If a new message has to be put on the blackboard as a result of the collected message, the *messageHandler()* also takes care of that.

The third member of the class Agent is *inferInfo()*. It constantly tries to derive new information from the agent's knowledge base and available data. This could concern facts, undefeasibles, preferences or normalities, but only about the user. The rules and facts about the diary and todo-list are static. Since user-agent is the only one that learns, all newly discovered information has to be sent to him, so that he can add, remove or augment the appropriate formulas of the user model. The final method they inherit from the Agent class is *sendInfo()*. This one is called when either the diary-agent or the todo-agent has discovered something about the user. The new information is sent to the user-agent so that he can add it to his knowledge base, hopefully increasing his resemblance to the human user. The user-agent does not inherit the *sendInfo()* method. He has no use for it, because the other two agents do not learn. See Sections 3.3 and 4.6 for some thoughts on learning techniques.

As stated in Section 3.1.3, OutOfBrain and other graph rewriting systems do not yet possess modularity in the way that object-oriented programming languages do. Therefore, inheritance is not an integral part of the architecture. It can, however, be simulated. Thinking of the three agents as sub-classes of the super-class Agent aids us in the design process, even though we do not use true inheritance.

4.4.3 Overview of data

Before we turn to the description of the chosen methods for communication and learning, we have to devote some attention to the data-members of the three agents. Below, we treat each of the data-members occurring in Figure 4.4 in some detail.

User-agent

The user-agent has access to the current view of the diary and todo-list (*currentDiaryView* and *currentTodoView*). He can also remember or see the last couple of entries in the conversational text-interface (*diaryConversationHead* and *todoConversationHead*).

Determining exactly how much of the conversation history he should remember is tricky. As can be seen from Figure 4.1, the conversation history is divided into two parts. One of them contains the messages to and from the diary-agent, the other contains the ones to and from the todo-agent. A straightforward choice would be for example to make the last six entries in both parts available to the user-agent. But most of the time, the user is having a relatively long conversation with one of the agents. Meanwhile, he or she forgets the previous conversation with the other agent. For the user-agent to be a good representation of the human user, we need to determine the available entries of the conversation histories dynamically.

The user-agent maintains a model of the human user in three separate arrays (*userPreferences*, *userNormalities* and *userUndefeasibles*). We mentioned earlier that there is a fourth kind of information, namely facts. These are also stored in the array of undefeasibles. To model the user more accurately, the user-agent also has four data-members to store additional information. First of all, there is the member *distanceMatrix*. It is the graph equivalent of a two-dimensional array that contains the travel times between locations. Secondly, there is the member *typeTree*. Here, the types and sub-types of the user's activities are stored, so that the user can learn their typical attribute values (normality rules). The third type of additional information in the user model is called userEnergy. It tells the user-agent how much energy the user has on an average day. Finally, the user-agent maintains the data-member *defaultWeek*. Here, information is stored about the typical week for the user. It includes normal work times and obligations that return every week. These should also be treated as normality rules, because of their defeasible nature.

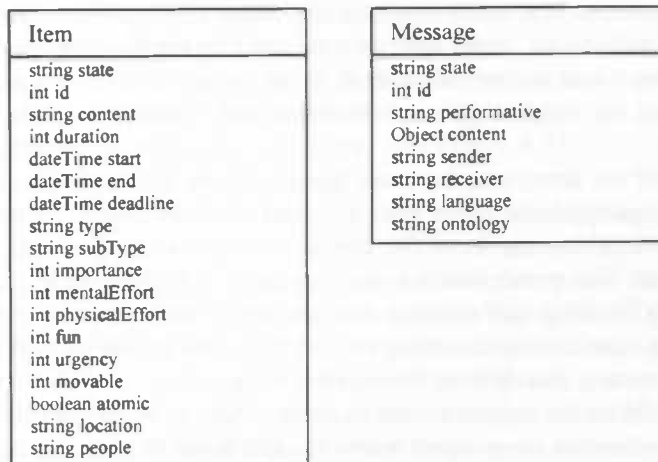


Figure 4.5 The data-classes Item and Message.

Diary-agent and Todo-agent

The diary-agent has access to the entire diary and its history. It represents about two years of the user's life, moving along with the current date. The full history of messages that were sent and received by him are also at his disposal. Furthermore, he maintains three knowledge arrays of his own. These contain information about what are strict rules and preferable or normal situations for a diary. The todo-agent's action and perception radius is similar to the diary-agent's. He maintains the todo-list and has access to its history. Furthermore, he has access to his own conversation history and three knowledge arrays.

Neither the knowledge arrays of the diary-agent or the todo-agent are subject to learning. When one of them derives user-specific information from his conversation history and diary/todo-list history, he sends it to the user-agent instead of storing it in his own

knowledge base. Many of the data-members in Figure 4.4 were especially designed for the diary assistant. Two of these newly defined data-types, *Item* and *Message*, are quite complex. They are explained in detail below (Figure 4.5). The remainder of these newly defined types, *Day*, *Preference*, *Normality* and *Undefeasible* are just strings that obey certain constraints. They do not require any further explanation.

Both the Item and the Message class have a *state* and an *id* (identifier). The state is used to store information about the situation an item or message is in. For example, an item can be waiting for alternative dates and times and a message can be waiting to be picked up from the blackboard. The id makes for a convenient way of referencing.

The Item class

This class serves to represent todo-items and diary-items of many forms. Most attributes of this class are optional. The user does not have to specify everything; he or she can for example add a todo-item without a deadline. First of all, there is the *content* of the item. This variable can hold strings like "phone John" or "staff meeting". The *duration*, *start*, *end* and *deadline* attributes represent the temporal aspects of an item. These are used during planning. The diary-agent should apply preferences like "no suggestions beyond the deadline" and "suggest alternative times close to the desired time of a conflicting item".

The todo-agent also utilises these temporal aspects. A todo-item that takes only fifteen minutes for example will be moved to the diary before items with a long duration. The attributes *duration*, *deadline*, *importance*, *mentalEffort*, *physicalEffort* and *fun* determine in a weighted sum which todo-item (group three) will be sent to the diary first (*urgency*). We call this a weighted sum, but the calculation is actually implemented using QDT-preferences. This is the most important task of the todo-agent. Items that were rejected by the diary-agent due to overlap in time (group two) are not sent to the todo-list. It is the responsibility of the diary-agent assisted by the user-agent to suggest alternative dates and times for these items immediately after rejection.

The *type* and *subType* of an item can be used by the team of agents to obtain knowledge about the world and particularly about how the user likes to handle items of a certain type or sub-type. These variables can have the obvious values work, household and leisure, but also less obvious ones like administration and healthcare. Typical sub-types for work are, among others, meeting, writing and reading. Leisure could be divided into sports and relaxation. A diary assistant that is truly adaptive should have the ability to learn new types and sub-types, because every user has different types of activities.

The attribute *movable* indicates the degree to which a diary-item is allowed to move in time. This attribute is important when the diary-agent wants to shift items to make room for a new one. The attribute *atomic* tells us whether or not an item can be split into parts. This attribute is required when planning an item that takes many hours, writing a scientific paper for example. The system can help the user plan such items by proposing possible divisions. Finally, *location* and *people* hold information about where the activity takes place and with whom. Our team of agents can commence a dialogue with the user to find out how long travel times are between locations like home and office so that this can be used in planning. Knowledge about people can also be utilized in planning. For example, it pays off to know how flexible another person is in changing appointments. It is also worthwhile to remember how reliable and important people are.

The Message class

All communication takes place through messages. It is the only way information can be brought across. The *performative* indicates what type of message we are dealing with. In the theory of speech acts, each utterance has an intended effect [Searle 1969]. Possible effects

include the activation of another agent's methods or the communication of requested information. The attribute *content* contains the message itself. This can be a simple string, but also an entire todo-item along with a list of possible dates and times for it. Different performatives lead to a different interpretation of the content. A typical message from the todo-agent to the diary-agent would be a request to put an item on the diary. "Request" is the performative here, and the item with all its attributes form the content.

The *sender* and *receiver* attributes store the names of the agents involved in a message. These attributes come in handy when agents check the blackboard for new messages or when they are constructing messages themselves. Finally, the attributes *language* and *ontology* have been introduced to ensure a common frame of reference for a group of agents. We could use these attributes to guide the parsing of messages.

This concludes our discussion of the diary assistant's methods and data-members. The following Sections shed some light on our ideas for communication, learning and interface design. Then, we move on to Chapter 5, which is concerned with the implementation of the adaptive diary assistant in OutOfBrain.

4.5 Methods for communication in a diary assistant

The interface, the different parts of the environment and the agents are strictly divided. For the current application however, this is not really necessary. A single agent with access to the entire environment could do the job just fine. In fact, an adaptive diary assistant with only one agent would probably be easier to implement. In our design however, the agents cannot access each other's data or methods. Instead, they have to cooperate via messages on the blackboard. As stated before, the messages through which the agents communicate obey the FIPA standard [Rumbaugh et al. 1989] (see Figure 4.5).

We have chosen this approach, because we are interested in true MAS. Recall that our aim is not to develop a commercially interesting application. Instead, the diary assistant serves as a test-bed for OutOfBrain and QDT+. True multiagent systems are by definition distributed in space. OutOfBrain does not yet allow for spatial distribution, so we have to simulate it instead. The main difference between this simulation of distributedness and real distribution is that the three agents and the four parts of the environment are located on the same computer. In fact, they are even in the same file. We only pretend that the agents cannot access each other's data. We claim that from here, creating a true MAS would only be a matter of moving the agents to different computers and setting up safe communication channels between them.

To provide some insight in the way communication is used in the adaptive diary assistant, we focus on two of the many tasks of our multiagent team. The first is their main task: planning todo-items. It can be divided into three subtasks: choosing the most urgent todo-item, creating a list of possible dates and times for a todo-item and choosing the best date and time from this list. The second task we treat here is maintaining the distance-matrix. The user-agent uses this part of the user-model to store the locations of all items it encounters and the travel times between them. The system can use the distance-matrix to make sure that travel times between the locations of diary-items and the location of the todo-item are taken into consideration when choosing possible dates and times for todo-items. Furthermore, the distance-matrix can be used to minimize travel times during planning.

4.5.1 Communication during planning

The ability to move items from the todo-list to the diary is one of the most important factors that distinguishes our diary assistant from existing ones. Figure 4.6 shows the information flows required for this ability. Planning begins when one of the buttons on the interface is pressed, indicated by a hash-symbol.

When this button is pressed, the interface sends a *cfp* message (call for proposal) to the todo-agent. In effect, the interface asks the todo-agent to choose the most urgent item on the todo-list. Upon receiving this message, the todo-agent will compare the attributes of the items on the list and hopefully come up with the one that should be taken care of first. The ideal todo-item (the one that is to be moved to the diary first) is extremely fun and takes little to no effort. Furthermore, it should be important, the deadline should be close and the activity should take little time. The item that satisfies the most of these preferences will be chosen. Note that we only use preferences here. However, normalities could also be checked for satisfaction while choosing an item. Once this is done, the todo-agent will *propose* the chosen item to the interface and thus to the user.

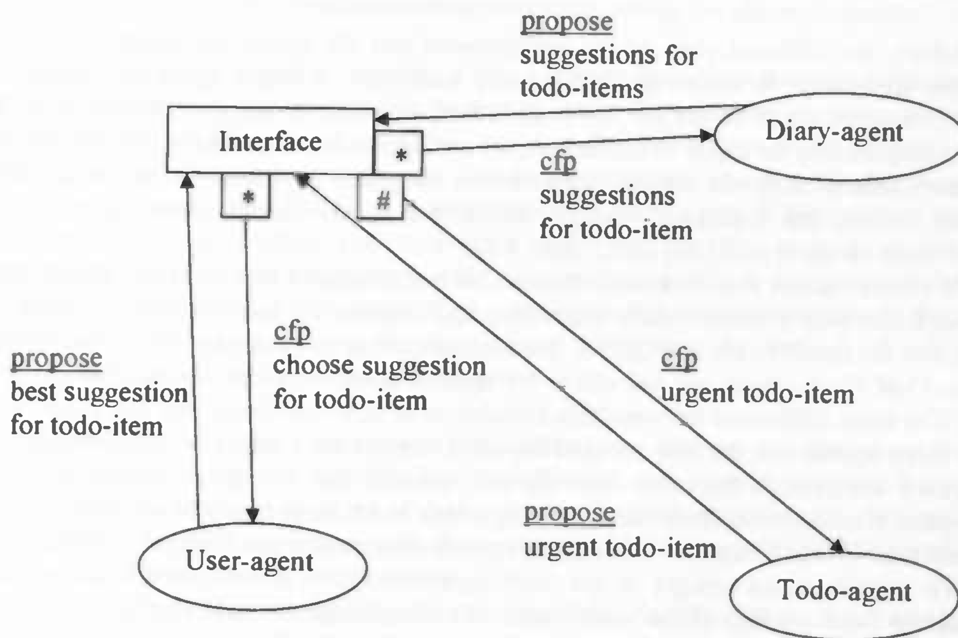


Figure 4.6: The message flows concerned with the planning of todo-items. The asterisks and hash-symbols indicate button-presses by the user.

After pressing another button on the interface, indicated by the asterisk-symbols in Figure 4.6, the user will be aided in moving an item from the todo-list to an appropriate date and time on the diary. This can be either the item chosen by the todo-agent or another one, chosen by the user himself. First, a message is sent from the interface to the diary-agent. Upon receiving it, the diary-agent will generate suggestions for the todo-item. He starts searching for unoccupied periods at the current date and time until some condition is met. That is, when the diary-agent reaches the deadline of the todo-item, it should stop collecting suggestions. Also, the diary-

agent should stop searching when a given number of suggestions has been collected (e.g. fifty). While looking for these unoccupied gaps, the diary-agent takes into account the travel times between the locations of the items already on the diary and that of the one to be placed. In order to do this, the system needs the distance-matrix to be up-to-date. If a required travel time is unknown, the user is prompted to provide it. See Section 4.5.2 for some more information on updating and using the distance-matrix.

The list of suggestions that results from this search is sent back to the interface as a proposal. The interface forwards this message to the user-agent as a cfp. His task is to select the best suggestion. Again, by the best one, we mean the one that satisfies the most preferences. Just like with choosing the most urgent todo-item, choosing the best suggestion can make use of normalities as well as preferences. However, we claim that in the current domain, preferences can yield rational decisions, even without the consideration of normalities. These are the preferences used to guide the choice: the user's energy threshold should be obeyed, travel times should be minimized, monotonous days should be avoided and user-specific preferences should be taken into account. The chosen date and time are proposed to the interface so that it can be highlighted for the user. To summarize, the planning skills of our adaptive diary assistant consist of three separate parts: Choosing the most urgent todo-item, generating a list of possible dates and times for a todo-item and choosing the best suggestion from that list.

4.5.2 Communication during distance-matrix maintenance

Another part of the system that makes extensive use of agent communication is the one concerned with the maintenance of the distance-matrix[1]. This part of the user-model is the graph equivalent of the kind of tables commonly used for distance or travel time lookup. The distance-matrix is used twice during planning. First, the diary-agent needs it when looking for gaps in the diary. The todo-item's duration as well as the travel times to and from the adjacent diary-items determine whether a gap is large enough. Second, the matrix is used by the user-agent when he chooses the best date and time from a list of suggestions. In particular, the choice is guided by the minimization of travel times. Consider for example, the situation where a meeting in Amsterdam has to be placed on the diary. Suppose one of the suggestions is to put it between two appointments in Groningen. The user-agent has a preference for minimizing travel times, so he will probably choose a suggestion for another day.

Of course, the distance-matrix needs to be up-to-date. For the input of new travel times, the system has to rely on the user. This is why we only want to update the matrix when a travel time is actually needed for generating suggestions or choosing a suggestion. Being prompted for travel time input is a little tedious at first. But after a while, as the distance-matrix becomes more complete, the user is rarely asked to provide a travel time anymore. Figure 4.7 does not show all communication concerned with maintaining and using the distance-matrix. It only contains the update mechanism initiated by the diary-agent while generating date and time suggestions. The user-agent can initiate a similar update mechanism, but this is not shown in Figure 4.7; neither is the actual use of the distance-matrix.

[1] The terms 'distance' and 'travel time' have the same meaning: The number of minutes required to get from one location to another.

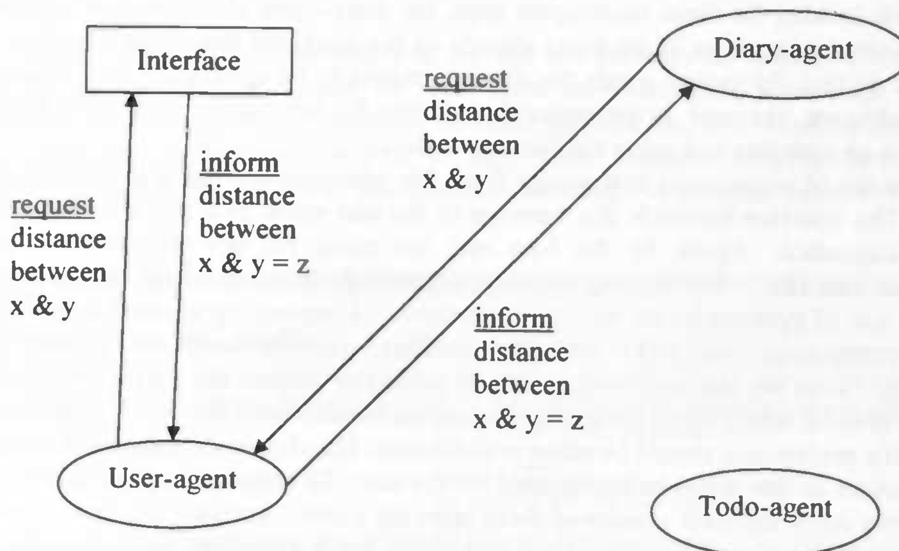


Figure 4.7: The message flows concerned with maintaining the distance-matrix.

When the diary-agent checks whether a gap between two diary-items is large enough for a certain todo-item, it needs to know the travel times between the locations associated with these items. Since the user-agent is the only one that has access to the distance-matrix, the diary-agent has to send him a *request*. The user-agent then searches the distance-matrix for the required travel time. If he finds it, he *informs* the diary-agent. If the required travel time is not present in the matrix, the user-agent will pass on the request to the interface. Consequently, the user is asked to provide the travel time. His or her answer is sent back to the user-agent in an inform message. This agent then stores the travel time in the matrix and passes the inform message on to the diary-agent. Either way, the diary-agent will receive the required information and can continue his work.

As can be seen from the information flows above, we do not use confirmation messages. Furthermore, items are moved instead of copied when sent to another agent. We do not need confirmation messages or copying of data, because the medium is safe; the system is not truly distributed. Whenever an item or part of the user-model is moved from one agent to the other, the responsibility to handle it moves along with it.

We refrain from describing all message flows in the system. First of all, there are simply too many tasks and sub-tasks to describe them all without losing the reader's attention. Second, most of the communication between the different agents is similar to the communication described above.

4.6 Methods for learning in a diary assistant

The ability to adapt to the user's needs is what makes our diary assistant special. The current Section provides an overview of the different learning techniques that are used. We refer back to Section 3.3 for a discussion of the modes of learning at our disposal. During reasoning, our agents employ three types of rules: undefeasibles, preferences and normalities. Learning amounts to creating, deleting and augmenting these rules as well as atomic facts. The rules

and facts deployed by the diary-agent and the todo-agent are static. The only agent that learns is the one modelling the user. As can be seen from Section 3.3, the ways the user-agent can learn things are diverse and plentiful. Below, we fill in these ways with concrete matters.

- **distance-matrix:** This learning task is concerned with maintaining a matrix of travel times between the locations of the user's activities. If the diary-agent knows for example that home and office are approximately fifteen minutes apart, he can take this into account when planning activities with the same values for their location attributes. The user has to communicate these travel times to the system explicitly when asked.
- **type-tree:** The user-agent can learn the meaning of the types and sub-types of items. He could for example derive which types of activities are relatively demanding for this user and then avoid planning more than one of these items on the same day. Also, the system should be able to add new types and sub-types along with information about their meaning.
- **user-energy:** The user's average amount of energy for a given day can be learned by the user-agent. This is done by statistical analysis. We distinguish between different types of energy. Mental energy is something else than physical energy. In fact, physical effort in sports can replenish one's mental energy.
- **default-week:** The maintenance of a model of the typical week for this user. The planning of new items by the diary-agent can use this information. For example, items of the type work should not be planned after five o'clock or during the weekend. The system should also be aware of facts such as "The user plays soccer every Wednesday night." and "He or she spends less energy on Sundays." The system can augment the model of the user's typical week using both implicit methods and explicit methods: Analysis of history or repetitive punishment could indicate that the typical week has changed. Alternatively, a dialogue initiated by the user or the diary assistant can be used for this learning task.
- **alternative dates and times:** The way the diary-agent collects alternative dates and times for items could also be subject to learning. How many should he collect and how big should the time-interval between two suggestions be. This depends on the item's duration, but also on user-specific preferences. A dialogue with the user can be initiated to obtain the necessary information.
- **level of occupancy:** By analysing the user's behaviour and the history of the diary and todo-list, we can maintain the level of occupancy of a day or week. This is used in planning, but also in adaptation. The todo-agent could for example derive from history that many todo-items are added on Wednesdays. He could ask the user to confirm this fact and then the user-agent could anticipate on it during planning.
- **search window:** The augmentation of the search window for collecting alternative dates and times for items. Some people like to spread their todo-items over a long period of time, while others like to get things done as soon as possible.

There are many more opportunities for learning in the domain of appointments and obligations, especially if the system is able to construct new preferences and normalities. Unfortunately, we did not have time to actually implement a general purpose method for constructing new preferences and normalities (using the modality time as well). Consequently, the application we ended up with does not have all the adaptation skills we intended it to have.

4.7 Interface design

As was already mentioned in Section 3.4, we did not perform an extensive study of the literature on interface design. Instead we drew upon our own factual knowledge as well as Rockingstone's experience with developing database applications. The resulting design obeys the Rockingstone interface-style and it is built from standard Delphi components. A screenshot is included in Figure 4.8. We trust that the design is intuitive enough to speak for itself.

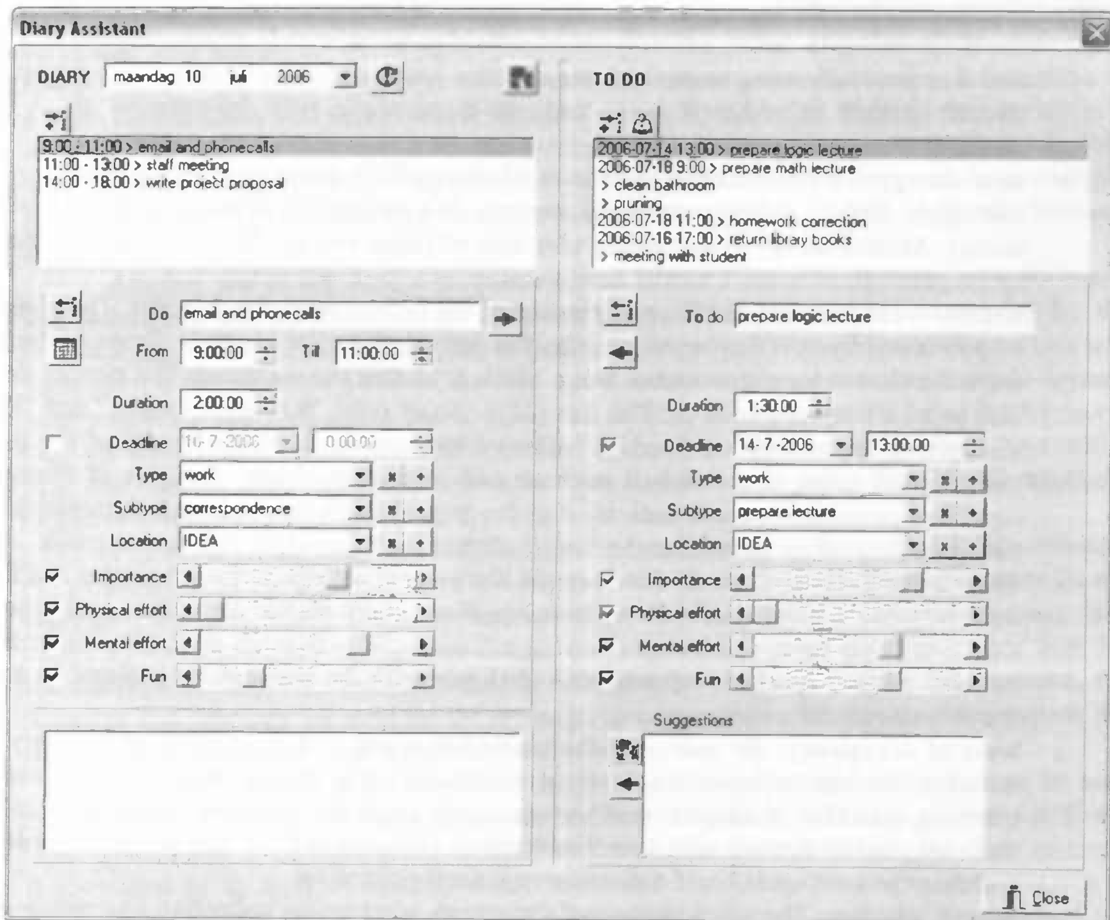


Figure 4.8: The interface of the adaptive diary assistant with the diary on the left and the todo-list on the right.

This concludes the current Chapter, but there are two more Chapters to go. Chapter 5 elucidates the implementation of the diary assistant and Chapter 6 consists of our conclusions and an extensive discussion on evaluation and future work.

5 Implementation

We use OutOfBrain for the implementation of the adaptive diary assistant. That is, the *view* and *control* components are implemented in Delphi, whereas all the *model* components are implemented in OutOfBrain[1]. Since we are dealing with a new programming environment that is still under development, the implementation phase is scientifically more interesting than is normally the case. We use an incremental strategy; the design changes during implementation where appropriate. Furthermore, the functionality of OutOfBrain itself is improved as needed. See Appendix A for the current version of the OutOfBrain manual.

Unfortunately, many of the ideas in our original design did not get implemented. Section 5.1 mentions the components that did not make it past the design stage. Some of the reasons and possible remedies for these shortcomings are discussed in Sections 6.1.3 and 6.3.1 respectively. The main reason for them however is simply the time pressure. Once we arrive at Section 5.2, we have a clear picture of what is actually implemented. Section 5.2 explains the implementation on an abstract level of description. It contains an overview of one of the three agents where each rewrite rule is represented as a single node. Overviews of the other two agents can be found in Appendix B. In Section 5.3, a small part of the implementation is explained in detail. The reader is encouraged to consult Appendix C for some host graph representations. Section 5.4 focuses on the use of preferences to guide decision making[2]. Finally, to show that the diary assistant actually makes intelligent decisions, we include a small test report in Section 5.5.

5.1 Design reconsideration

The design presented in Chapter 4 is not implemented in its entirety. Instead of changing the design Chapter to fit the implementation, we choose to leave it unaltered, because we would like to leave the complete design for future implementation. We just did not succeed in implementing it all. The parts of the design that did not get implemented are presented here and they are discussed a bit further in Sections 6.1.3 and 6.3.1.

There are two shortcomings of the actual implementation that are quite regrettable from a scientific point of view. The first is that we did not use normalities in our adaptive diary assistant. We did make the functionality for normalities available in OutOfBrain, but we did not use them in the adaptive diary assistant. The second regrettable shortcoming is that we did not incorporate time, the third modality of the QDT+ architecture, into OutOfBrain.

[1] The model, view, control(ler) architectural pattern dictates a strong distinction between the three main components of an application. Modifications to one component can be made with minimal impact on the others. The model component contains everything that happens on the inside, whereas view and control are the parts that take care of interaction with the user.

[2] One of the most important deficits of the implementation is that we do not use normalities. Of the two CO-models that make a QDT-model, we use only the one for preferences.

Consequently, we could not use it in the adaptive diary assistant either. Implementing the QDT+ design presented in Chapter 4, including normalities and time, is something for potential future work.

Partly because the modality time did not take solid form, we did not have the means to implement a general purpose learning method either. The plan was to create a mechanism that analyses history in order to add, remove and change preferences, normalities and undefeasibles. Instead, we ended up with a system that is much more static than the one we set out to create. The system does learn, but only in a predefined, highly constrained problem space that fails to impress.

Besides these major design reconsiderations, a couple of other ones have taken place, especially with respect to which agent does what. We found that (simulated) distributedness forces one to think carefully about who does what, because communicating information to and fro is both costly and complicated. Section 5.2 exactly describes the task distribution as it was implemented. Furthermore, from the methods for learning presented in Section 4.6, only the top three were actually implemented: distance-matrix, type-tree and user-energy[1]. Furthermore, most communication now runs via the user-interface instead of directly between the agents.

Another one of our design reconsiderations is that during the planning of todo-items, the user cannot ask for a new set of dates and times. When the user is not satisfied by the chosen suggestions, the re-planning has to be done manually. Yet another component that was withdrawn is the supporting role of the user-agent and his user-model during the todo-agent's item selection. The todo-agent has to make do without the user-agent's help. Furthermore, only the human user can decide that it is time to move a todo-item to the diary. Our initial plan was that the diary-agent and todo-agent can also decide so when the diary is relatively empty or when the todo-list is relatively long.

The items on the diary and todo-list also had some components removed. In particular, the attributes *atomic* and *people* were not implemented (Section 4.4.3). The same holds for the message class's members *language* and *ontology*. We feel that the removal of these four attributes was a good move. The user already has to provide a lot of information per item for the intelligence to work, even without the attributes atomic and people. Furthermore, the language and ontology we used for the messages is unequivocal. The agents always speak the same highly constrained language, so the attributes are obsolete.

Also with respect to the interface, things have changed considerably. As can be seen from Section 4.7, we do not use a conversational text-interface[2]. We had to choose a different input method due in part to time constraints on the project. Instead of text, the diary assistant is controlled by an array of buttons, check-boxes and text-fields. We know from research in language processing that a conversational text-interface is only useful if it is worked out properly, even if it uses a formal syntax. Obviously, we did not have the time for that. However, Rockingstone does intend to incorporate conversational text-interfaces into future applications.

The following Section contains an overview of the diary-agent, represented in the OutOfBrain equivalent of pseudo-code. Similar overviews of the todo-agent and the user-agent can be found in Appendix B. The style of the Figures in Section 5.2 and Appendix B should be familiar. OutOfBrain is also a convenient tool for creating diagrams, so we decided to use it for the implementation overviews.

---

[1] User-energy is called average-energy in the implementation. Consequently we use both these words to refer to the same part of the user-model. Recall that we also use distance and travel-time interchangeably.

[2] Chapter 4 describes the old design, whereas Chapter 5 relies on the new design. Section 4.7 is an exception, since it describes the interface that was actually implemented.

5.2 Process overview

Without further ado, we now present a substantial part of the implementation. It would be nonsense to provide all the code itself, so we have translated the three agents to the OutOfBrain equivalent of pseudo-code. There are three processes in total, one for each agent. In this Section, we only provide the diary-agent. The other two can be found in Appendix B (without any further explanations). For sake of simplicity, some parts of these overviews have been simplified or changed a bit. In particular, we do not always stay true to the edge labels connecting the rules (before/after, implied, true/false, see Appendix A, Chapter 5). These are minor details however; functionality is identical in implementation and overview. Please consult Appendix C for some coverage of the host graph.

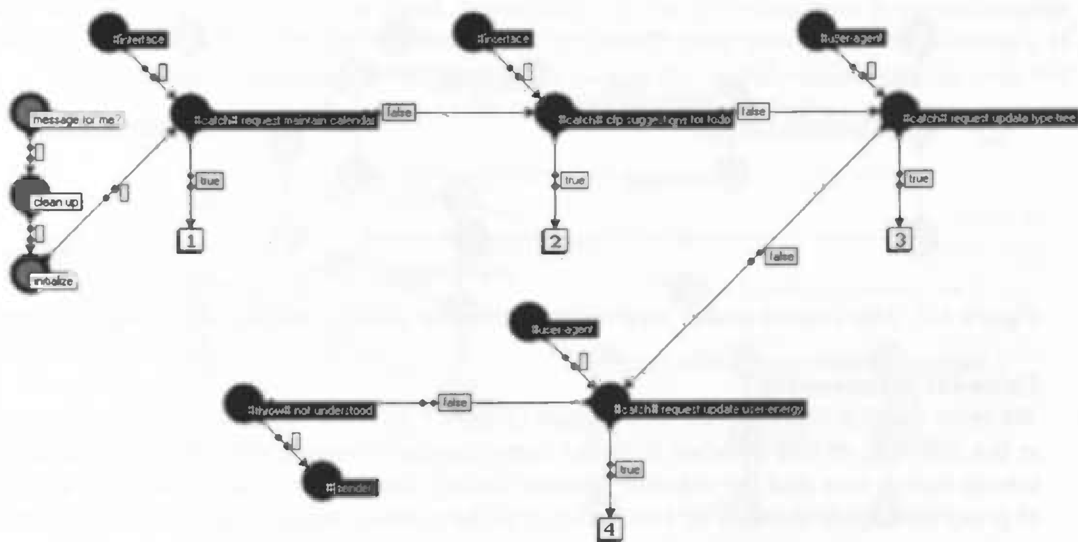


Figure 5.1: The diary-agent's skeleton. When a message is caught that satisfies certain constraints, one of the rules labelled '#catch#' returns true. Activity then moves from this skeleton to one of the main branches depicted in Figures 5.2, 5.4a, b, 5.5 and 5.6.

The user-agent
Like the other two agents, the user-agent begins each iteration with checking whether there is a message on the blackboard with receiver 'diary-agent'. If this is the case, old references are cleaned up and a new one is created. See Section 5.4 for a detailed description of some rewrite rules, including clean up and initialization. Once this is done, the message's attributes are matched against four different moulds. Each one serves as a door to the main branches of the process. The first one catches messages of the form "request maintain calendar". Such messages are sent by the interface whenever the calendar has to shift forward in time. We want to have one entire year ahead of us in the diary at any given time. If the mould fits the message, activity moves down to the square labelled '1'. This same square can be found in Figure 5.2. That is where the process continues. In reality, the user-agent consists of a single process, but we could just as well distribute the parts depicted in Figures 5.2, 5.4a, b, 5.5 and 5.6 over different processes, different graphs or even different computers. The numbered squares could then act as interface places (see Section 3.1.4 and 3.1.5).

If the message found on the blackboard is not of that form, we move on to the next mould. This one checks whether the message is a call for proposal on suggestions for a todo-item. Such messages are sent by the interface whenever the appropriate button is pressed by

the user. If the condition returns 'true', activity moves to Figure 5.4 via interface place 2. The diary-agent should then create a list of suggestions for the todo-item that is highlighted on the interface. When this is done, the user-agent has to choose the best suggestion according to an array of preferences. The communication involved in this triangular cooperation is explained in 4.5.1. For the user-agent's role in this task, we refer to Appendix B.

Figures 5.5 and 5.6 can be entered through places 3 and 4 respectively. These branches update both the type-tree and the user-energy (see Section 4.6 and Appendix C). The distance-matrix does not need such an update mechanism, since it is maintained automatically whenever a new travel time becomes available. If none of the moulds fit the message, the diary-agent throws a "not understood" message back to the sender. Below, we elucidate these four branches of the user-agent.

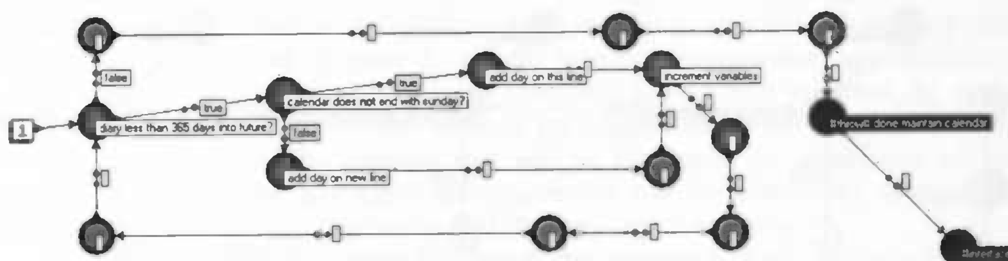


Figure 5.2: This branch makes sure that the calendar always reaches a year into the future.

Calendar maintenance

We enter the first branch of the diary-agent (Figure 5.2). This means that the message selected at the first step of this iteration is of the form "request maintain calendar" (Figure 5.1). This branch makes sure that the calendar always reaches one year into the future. The easiest way to grasp the Figure above is by comparing it to the corresponding pseudo-code in Figure 5.3[1].

```
while calendar less than 365 days into future do
  if calendar not ends with Sunday
    add day on this line;
  else
    add day on new line;
  increment variables;
end do
throw done maintain calendar;
```

Figure 5.3: Pseudo-code equivalent to Figure 5.2.

Suggestion generation

If the selected message is of the form "cfp suggestions for todo" instead, activity moves to Figure 5.4 via interface place 2. This branch is much more complicated than the previous one. It is responsible for planning todo-items. Because the branch is so complex, several clean up and initialization steps need to be done at the start of each iteration. The main loop of the branch starts by checking whether there are already ten suggestions for the todo-item. If so, the user-agent is done and can proceed to sending the reply message with the gathered suggestions attached to it. If not, the user-agent enters the loop by checking whether some

---

[1] In fact, the method of representation used in Figure 5.2 is an OutOfBrain equivalent of pseudo-code.

search-variable[1] has reached five o'clock already. If this is the case, the search-variable moves to eight in the morning of the following day and activity shifts along the edge labelled "implied" (see Appendix A, 5.6). After another clean up and initialization, the sidetrack joins the main loop again at the rule that checks whether the search-variable has passed the deadline. Obviously, an agent that suggests dates and times past a todo-item's deadline is a bad planner. If the search-variable has passed the deadline, and the user-agent collected five suggestions already, he is done and breaks from the loop. If not, he needs to come up with at least one more suggestion.

Producing a suggestion starts with a set of overlap checks. Items on the diary can have overlap with a suggestion in one of three ways, rendering the suggestion useless. The item on the diary could envelop the entire suggestion, from start to end ('overlap0?'). In this case, the search-variable and the search-variable plus the todo-item's duration are both after the do-item's start and before the do-item's end. It could also be the case that there is partial overlap at either the front end side or the back end side ('overlap1?' and 'overlap2?' respectively). If any of these checks returns 'true', the user-agent makes the search-variable jump over the conflicting do-item and then loops back to the beginning of the branch.

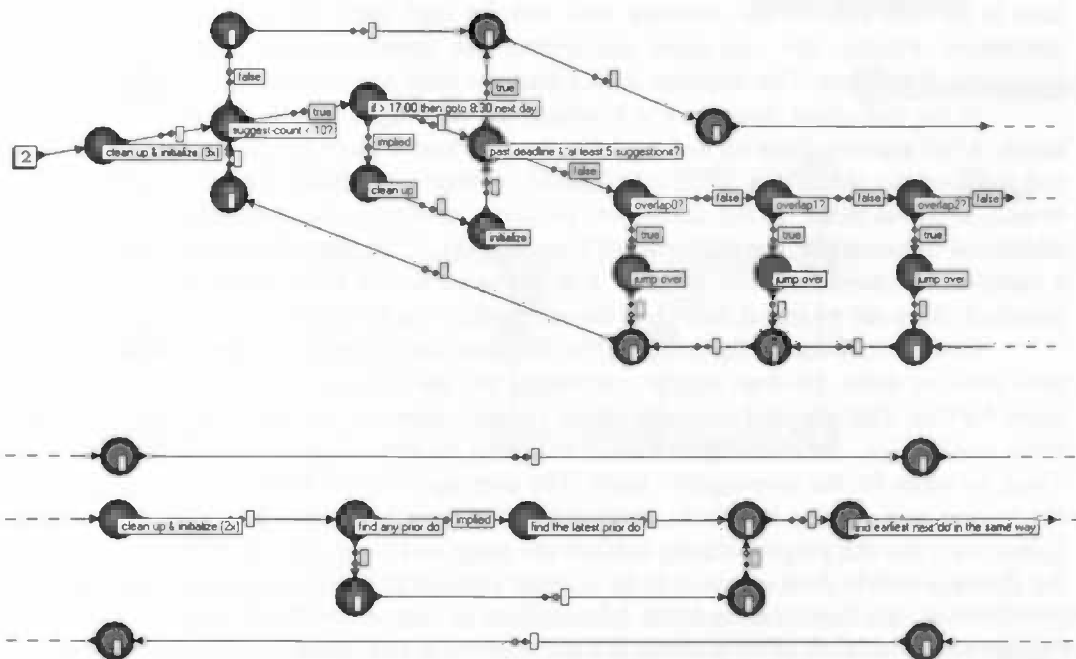


Figure 5.4a: First half of the diary-agent's second branch that generates a list of suggestions for a given todo-item.

Only if no overlap occurs, can the current suggestion be a good one. But before the user-agent accepts a suggestion, he has to take into account potential overlap caused by the travel times between the todo-item and the adjacent do-items. Furthermore, he has to store these do-items in the reply-message, so that the user-agent can make an informed decision as to what is the best suggestion. In particular, the user-agent tries to choose a suggestion that leads to

[1] This search-variable is a date/time object that keeps track of where, or rather when, the diary-agent is currently searching. It is initialized to hold the current date and time (taken from the system clock). During suggestion generation, it jumps half the todo-item's duration after finding a good suggestion and it jumps over a do-item when overlap occurs.

alternation of subtypes and attributes. The first thing the user-agent has to do is find these adjacent do-items. He does this by first selecting one from the appropriate day at random. Then, the one just before and just after the suggested time are found through pairwise comparison. Once we have found these so-called "latest prior do" and "earliest next do", we can proceed to Figure 5.4b.

Suggestions obeying travel times

The diary-agent only has to take travel times into account if the todo-item actually has a location. If not, the suggestion should be accepted and stored in the reply-message (after another clean up and initialization). The prior do-item and next do-item are also stored in the message, so that the user-agent can analyse them while choosing a suggestion. Furthermore, the travel times (distances) for the suggestion are sent along if it has any, because the user-agent tries to minimize these. To make the user-agent's job even easier, the diary-agent performs some preliminary calculations for him. He calculates the absolute differences between the attributes of the prior do and the todo-item (importance, mental effort, physical effort and fun). He does the same for the todo-item and the next do. Then the suggestion is done and it is glued to the todo-item between the suggestions already hanging there. The todo-item is in turn part of the message that will be sent upon the completion of suggestion generation. Finally, the user-agent increments the search-variable and starts again at the beginning of the loop. The step size is half the todo-item's duration.

If the todo-item does have a location, the user-agent could still be a long way from home. After another clean up and initialization, he would have to check whether the prior do and the next do also has a location specified. If these are absent, nothing can be done with respect to travel times, so the user-agent proceeds to storing the suggestion along with all its additional information (top-right part of Figure 5.4b). If the prior do or the next do does have a location, we need to check whether it is the same as the todo-item's location. If they are identical, there are no travel times and the suggestion can be stored.

However, if one of the locations is different, the diary-agent needs to know the travel time between them. He does not have access to the user-model, so he has to consult the user-agent for this. The required communication is represented by the two triangular formations of black nodes. First, the diary-agent throws a request for the distance between locations x and y. Then, he waits for the user-agent's reply. The user-agent might have to pass the request on to the human user via the interface, or the travel time might already be in his distance-matrix. Either way, the diary-agent simply catches the reply and stores the travel time. Consequently, the distance-matrix does not need to be updated periodically, because it is always up-to-date.

Now, we have reached the bottom half of Figure 5.4b. If there is a travel time (distance) between the prior do-item and the todo-item, the agent has to increment the search-variable appropriately ("jump over distance"). For the travel time between the todo-item and the next do-item, this does not apply. However, in both cases, the message has to be deleted and the user-agent has to check if the travel time is greater than zero. If not, overlap due to travel times cannot occur, so the agent simply moves on to storing the suggestion. In most cases however, the travel time will be greater than zero, potentially causing overlap. If it occurs, the suggestion is no good and the user-agent throws it away. He then moves back to "jump half a duration" and from there back to the start of the loop to create a new suggestion. If overlap does not occur, the user-agent stores the suggestion in the usual way and then he enters the loop again.

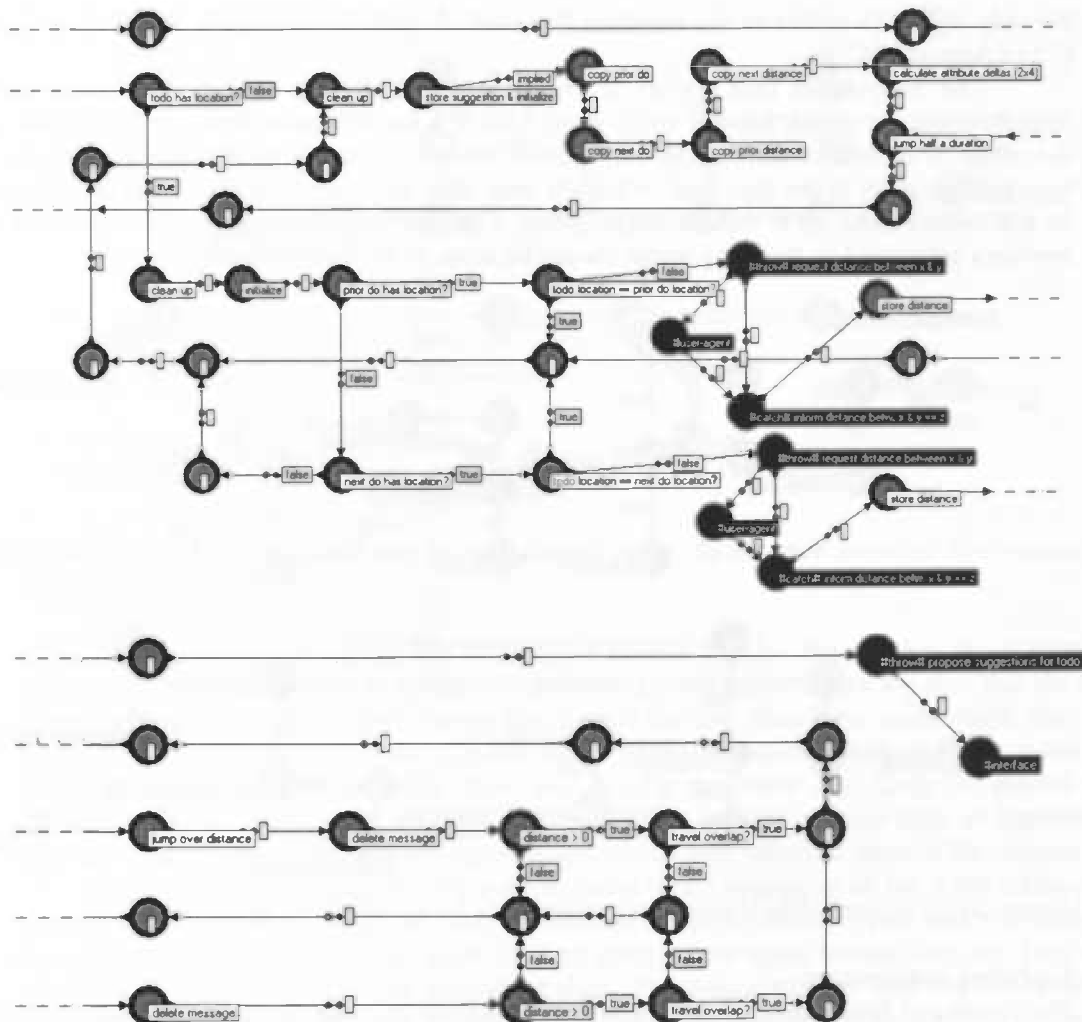


Figure 5.4b: Second half of the diary-agent's suggestions generator.

Updating type-tree

The diary-agent enters the third branch from Figure 5.1 if the selected message is a request to update the type-tree. It starts with a clean up, an initialization and the deletion of the old type-tree. Then, the process stays in a while-loop until all do-items have been analysed for the new type-tree. The types, subtypes and attributes of todo-items are not analysed, because this would only complicate the system with extra communication. Besides, every todo-item will eventually become a do-item that will be analysed for the type-tree.

An iteration through the loop starts with marking a do-item and then two clean up steps. Then, the diary-agent adds the marked item's type and subtype to the type-tree if they are not already there. Next, the attribute sums and counts associated with the item's subtype are incremented for all four attributes. When the diary-agent breaks from the while-loop, the sums contain the summed attribute values of all items of the same type and subtype. The counts are used later for dividing the sums into average attribute values. The item that was just analysed is marked as done and a new item is selected. This process continues until all do-items are marked as done. It is the diary-agent who does the updating, but the user-agent is

the only one with access to the resulting type-tree. A typical type-tree is depicted in Figure C.5 of Appendix C.

The diary-agent then arrives at the bottom half of Figure 5.5. After some more housekeeping, he enters another while-loop. This one has the same anatomy as the one just described. It calculates the average importance, mental effort, physical effort and fun for all type/subtype pairs in the type-tree. When the user adds an item of the same type and subtype, its attributes will be set to these average values. The user can still augment the values, but the averages calculated by the diary-agent should be close to the desired values.

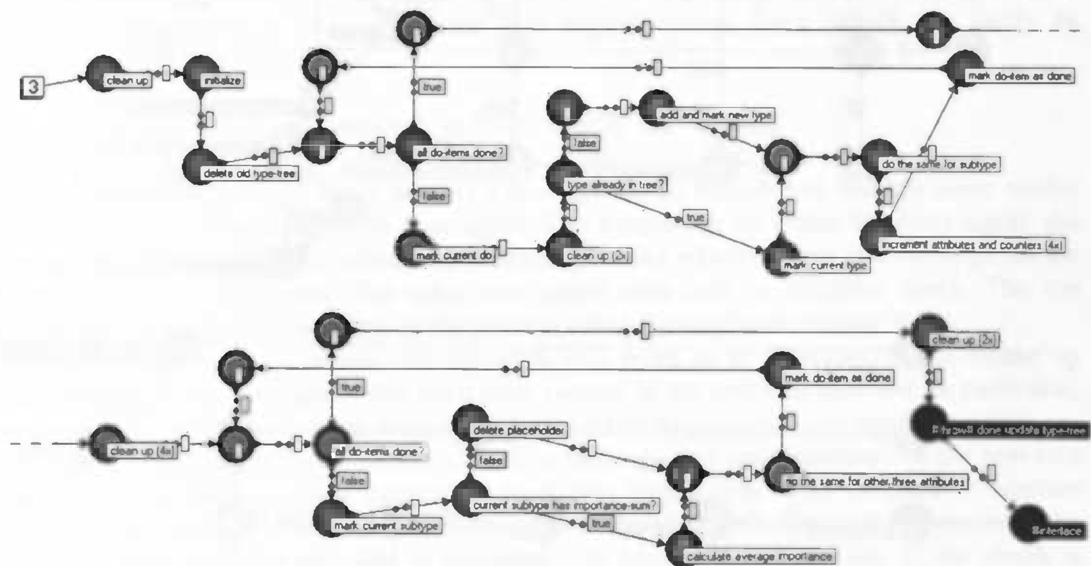


Figure 5.5: The part of the diary-agent responsible for updating the type-tree.

Updating user-energy

The fourth and final branch of the diary-agent processes messages of the form "request update user-energy[1]" (Figure 5.6). The user-agent needs this user-energy to be up-to-date because it is an estimate of how much mental and physical effort the user can handle on an average day. The diary-agent simply analyses the do-items on each day and calculates the average amount of effort per day (todo-items are not analysed). This measure becomes more and more reliable as the system is being used.

While the user-energy is not up-to-date, the diary-agent has to keep iterating through a loop. The user-energy is only up-to-date if every day up to yesterday has been analysed. So if this update process is called once a day, it only needs to go through the loop once. It starts by moving one day ahead from the day that was previously analysed. Then, after some housekeeping, the diary-agent checks whether there is an item on that day. If so, the number-of-days variable is incremented so that dividing the total amount of energy by the number of analysed days yields the correct average.

[1] We use the terms average-energy and user-energy to refer to one and the same thing.

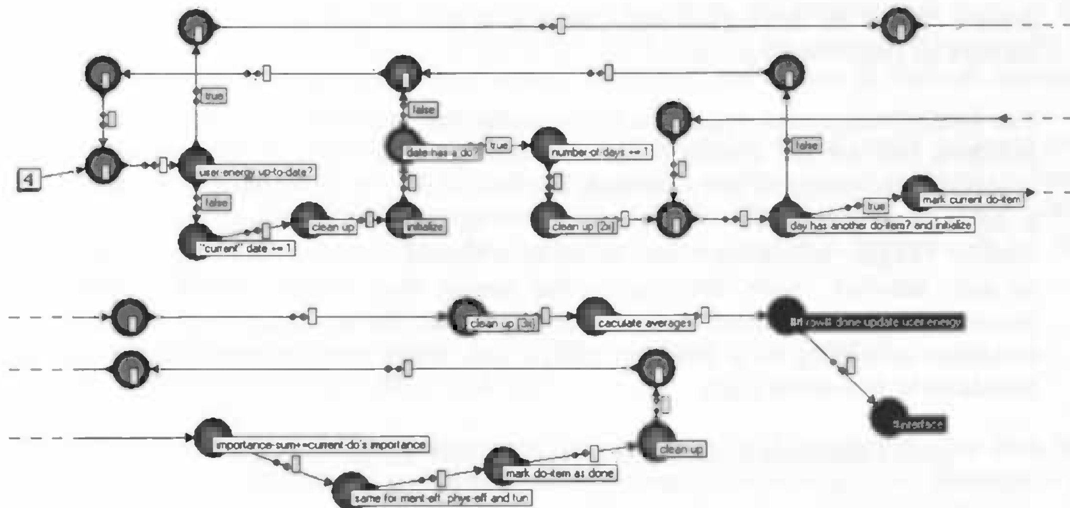


Figure 5.6: The branch that has to be followed when the selected message is a request to update the user-energy.

After some more housekeeping, the diary-agent checks whether the day has more items. If not, the day will not be used to update the user-energy, because chances are slim that the user spent the majority of his or her energy that day. If the day does have more items, they are analysed one by one in a nested while-loop. These sums then reflect the amount of importance, mental effort, physical effort and fun for the entire day. Only the mental and physical effort are used by the user-agent, but since it is so easy to keep track of importance and fun while you are at it, the diary-agent might just as well calculate them in the process.

When the day is done, the diary-agent starts at the beginning of the outer while-loop again. If there is another day to analyse, number-of-days becomes two and the analysis repeats itself. When there are no more days to analyse, the agent breaks from the loop and divides the attribute sums by the number of days. The user-agent can then use the resulting averages as mental and physical energy thresholds when planning todo-items. If adding a todo-item to a day results in violating either the mental or the physical energy threshold, the item had better be placed on another day. Unfortunately, the preferences that lead to this behaviour were not implemented, because they would require some extra data manipulation. We did not find the time to build these preferences and everything required for them.

This concludes our explanation of the diary-agent's four tasks. The todo-agent and the user-agent can be found in Appendix B. There is no text accompanying it, because we trust that the explanation of the diary-agent above makes the use of OutOfBrain-pseudo-code sufficiently clear. Consequently, the Figures in Appendix B should be readable without further elucidation. We now move on to some detailed descriptions of a couple of rewrite rules.

5.3 Some rewrite rules explained

An OutOfBrain program is essentially a host graph and a set of parallel processes, three in this case: diary-agent, todo-agent and user-agent.. These three processes are explained on a global level of description in Section 5.2 (diary-agent) and Appendix B (todo-agent and user-agent). Now, we take a closer look at part of the user-agent in order to get a thorough understanding of the inner mechanisms at work. We choose the part where every iteration through the

process (that is the user-agent) starts, because it gives a clear picture of how activity can flow through an OutOfBrain process.

The OutOfBrain search engine looks for nodes that are labelled <process>. Each such node indicates that we are dealing with a separate process that runs in parallel with all other processes. The adaptive diary assistant consists of only three of these, but theoretically there is no limit to the number of processes. Each of these nodes should have an outgoing edge labelled <begin> indicating where an iteration should start. For each begin-edge, there is also an edge labelled <run>, determining the current state of the process. In Figure 5.7, the process-node can be found in the upper-left corner. Its begin-edge and run-edge point to a condition consisting of a group of nodes with edges between them, which can easily be translated to first-order logic.

$$\exists x \exists y\ (Message(x) \wedge Sender(x, y) \wedge Receiver(x, user\text{-}agent))$$

This condition returns 'true' when there is a message on the blackboard that has some sender and a specific receiver, namely the user-agent. The sender can be either the diary-agent, the todo-agent or the interface. In effect, the condition checks whether there is a message on the blackboard for the user-agent. The other two agents start with an identical check. The run edge can only point to the next step in the process when the condition returns 'true'.

Once the condition is met, the run-edge will point to an implication that cleans up potential edges in the host graph that have been created in the previous iteration. In particular, it removes all occurrences of the graphical equivalent of *Message(ur-useragent, x)* where *ur-useragent* is a unique reference object and *x* is a message that was processed in the previous iteration[1]. There are two edge emanating from this implication. One of them is labelled <implied> and points to the same location again. This edge makes sure that whenever the implication fires and thus removes an occurrence of *Message(ur-useragent, x)*, the check is repeated. Only when no more occurrences are left in the host graph, the activity can move along the other emanating edge, labelled <next>.

Then, we arrive at the next implication, which creates an edge of the kind that was just removed. It looks for the unique node labelled [ur-useragent] and for a node labelled [message]. The <reference>-edge that runs from this message-node to the condition at the start makes sure that the implication chooses the message that was previously selected from the blackboard. Thanks to the creation of this pointer, the user-agent knows which message it is processing. He does not have to search for it again, which decreases the time complexity of the system.

Once the clean up and initialization are done, we arrive at a fork in the process. This is where the user-agent will look at the selected message more closely to determine what to do. The run-edge now points to a condition that checks whether the selected message has a particular performative and content. The message should be a call for proposal on choosing a suggestion for a todo-item. In other words; someone (the interface) is asking the user-agent to use his preferences to select the best suggestion for a todo-item from a given non-empty list. A first-order logic translation is provided below.

$$Message(ur\text{-}useragent, message) \wedge Performative(message, cfp) \wedge$$
$$Content(message, todo) \wedge \exists y\ Suggestion(todo, y)$$

[1] Note that the predicate *Message(x, y)* is nothing more than a tool to remember which message the user-agent is currently processing, whereas *Message(x)* from the previous example means that object *x* is a message.

If the user-agent finds that the message is indeed of this form, the activity will move to the right to process the message. If the condition is not met however, the message is apparently of a different form. In this case, the activity moves downward and arrives at the next message handler.

This one checks whether the message is a request for the distance between two locations (travel time in minutes). The diary-agent regularly needs to obtain these travel times from the user-agent's knowledge base, because he needs to take them into account while generating possible dates and times for todo-items. The condition corresponds to the following sentence.

$Message(ur\text{-}useragent, message) \land Performative(message, request) \land$
$\exists x \exists y\, Content(message, distance(x, y))$[1]

If the condition is not met, we are dealing with yet another type of message. In this case, the activity moves further downward to the next message handler. However, if our message is of the type described above, we move to the right until we arrive at an implication that is a bit more complicated than the rules we have encountered so far.

It copies the two locations from the message to the distance-matrix. The OutOfBrain engine ensures that this only happens if the locations are not yet present in the distance-matrix[2]. The reflexive implied-edge makes sure that both locations are copied if necessary before the run-edge moves on. The textual equivalent we provide is a little different than one would expect, because of the use of function symbols[7]. Thanks to the rule $distance(x, y) = distance(y, x)$ and the reflexive implied-edge, location-edges are created from the distance-matrix node to both locations.

$\forall x \exists y\, ((Message(ur\text{-}useragent, message) \land Content(message, distance(x,y))) \rightarrow$
$Location(distance\text{-}matrix, x))$

[1] The correct translation of *Content(message, distance(x,y))*, without using functions is $\exists x \exists y\, (Content(message, distance) \land Between(distance, x) \land Between(distance, y))$, which is not a natural sentence at all. The use of the function *distance* makes for a convenient representation. However, the current version of OutOfBrain does not yet allow the use of functions. It would require hyper-graphs, in which edges between edges are allowed as well as edges between nodes [Blostein et al. 1995]. Furthermore, we would have to restrict the number of edges allowed, so that the use of a function would always map to exactly one object. Partial functions could also be implemented.

[2] When this implication fires, we know that we are dealing with a new location. Instead of looking up the travel time himself, the user-agent has to pass on the request to the interface. The human user is then prompted to provide the travel time.

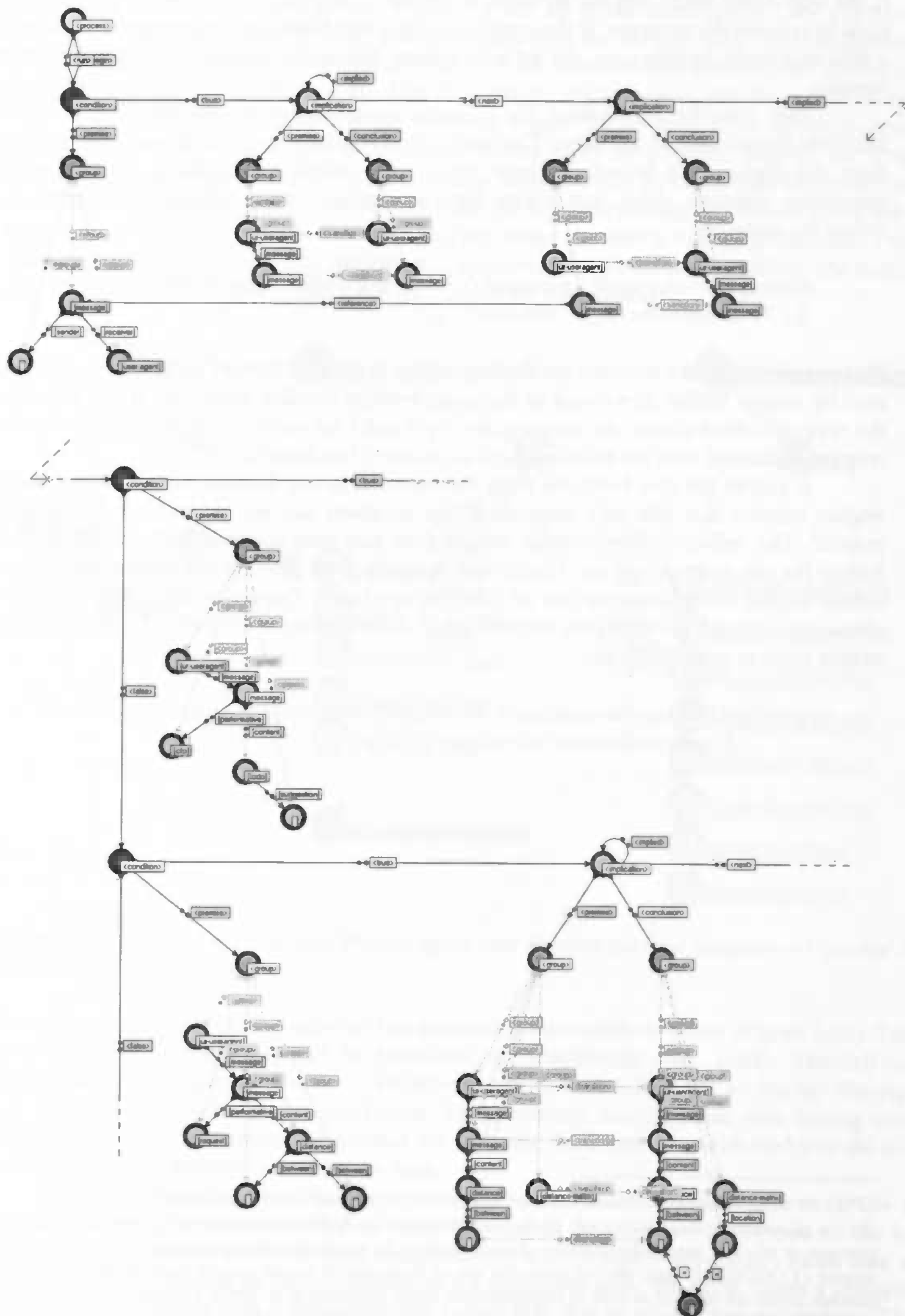


Figure 5.7: Part of the user-agent. The node labelled <process> and its <begin> edge indicate where an iteration starts. The <run> edge points to the implication or condition that is currently active.

5.4 Using preferences

In addition to implications and conditions, one can use so-called OutOfBrain-agents inside a process. These can use both preferences and normalities to guide decision-making. However, we found that the problem at hand can be solved with only preferences just fine. Both the todo-agent and the user-agent possess an OutOfBrain-agent that uses a collection of preferences. Below, we briefly explain them both since they are the backbone of the system's intelligence. Please consult Appendix A, Chapter 6 for a detailed manual to the use of OutOfBrain-agents.

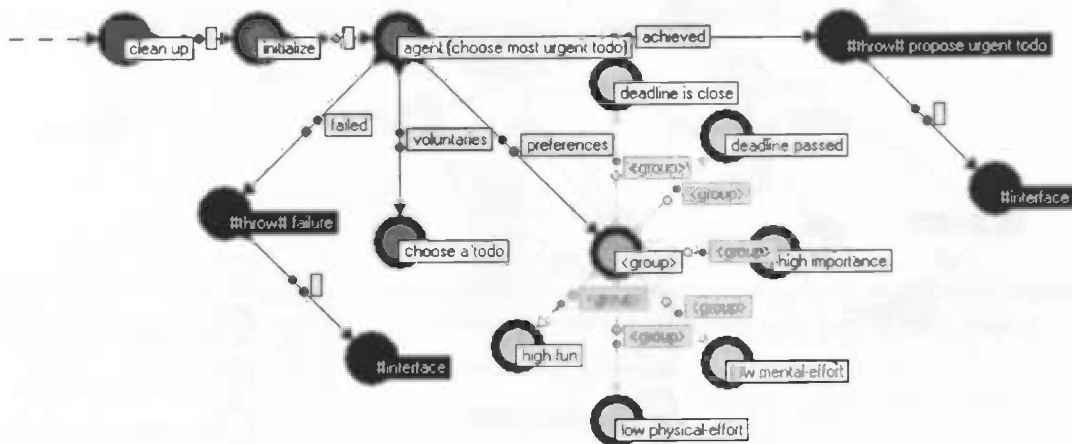


Figure 5.8: The OutOfBrain-agent that selects the most urgent todo-item on the list by satisfying as many of the six preferences as possible.

The todo-agent's OutOfBrain-agent is responsible for choosing the most urgent todo-item on the todo-list. It utilizes six preferences and it has only one voluntary action (see Figure 5.8). Choosing a todo-item as the most urgent one is nothing more than creating an edge labelled "most urgent todo" from a unique reference-node to the appropriate todo-item. What the agent should do if he does not succeed in choosing a todo-item is also specified: send the interface a failure message.

To understand why these six preferences lead to the selection of the most urgent todo-item, one should consider what attributes an extremely urgent todo-item would have. It would probably have high values for its importance and fun. This corresponds to the preference to do important things and fun things first. Mental effort and physical effort on the other hand should be low, because most people tend to postpone tasks that are mentally or physically demanding. Finally, if its deadline is close, an item increases in urgency. A past deadline also has a positive effect on an item's urgency. Whether this last one is a natural preference is debatable. We chose to omit a preference for items with a short duration, because the interpretation we use for mental and physical effort already contain this preference.

Inside the user-agent there is a similar OutOfBrain-agent that is responsible for choosing the best suggestion from a list generated by the diary-agent (Figure 5.9). Choosing a suggestion is again nothing more than creating some edge. And just like the todo-agent, if the user-agent fails for some reason, he needs to send a failure message to the interface. The eight

preferences on the right-hand side of Figure 5.9 are all of the same form. They express the user-agent's preference for attribute alternation and thus for diverse days. He maximizes the absolute differences between the four attributes of the prior do-item and the four attributes of the todo-item that is to be placed. He does the same for the todo-item and the next do-item.

The preferences on the left-hand side of Figure 5.9 express three different desires. First, the user-agent wants to plan the todo-item as early as possible. Second, he tries to interchange type/subtype pairs to ensure diverse days. Finally, he minimizes the travel times that come with the suggestion. Due to their need for some supporting functionality, we did not have time to implement preferences for staying under the user's energy thresholds. This is quite regrettable, since we did implement a sequence that periodically updates the required part of the user-model (see Figures B.6 and C.6 in the Appendices).

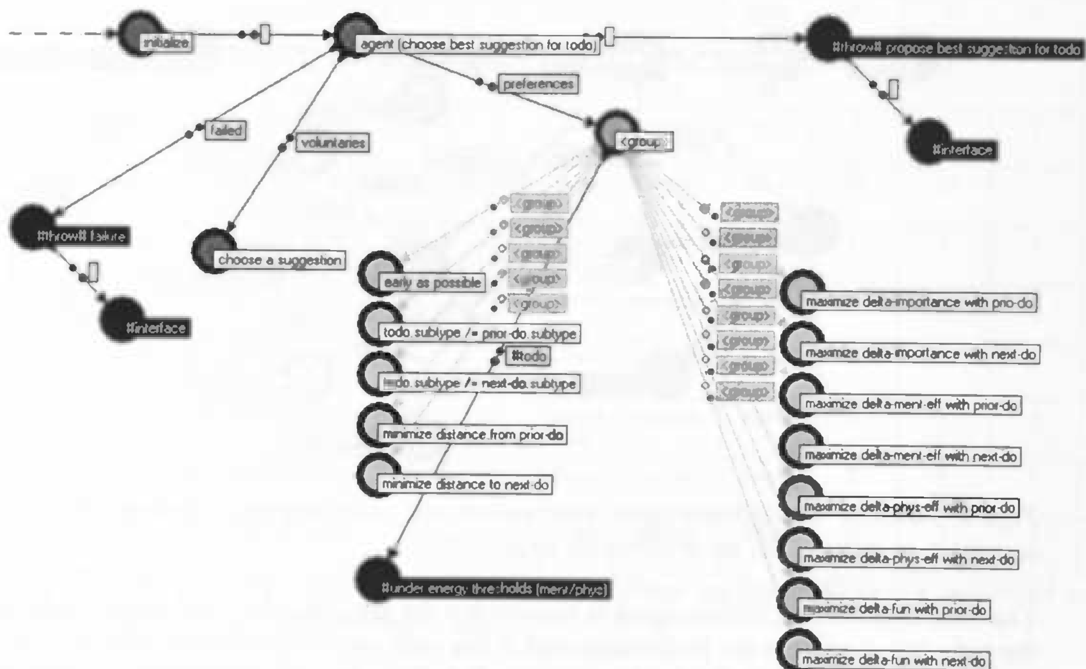


Figure 5.9: The user-agent's OutOfBrain-agent that chooses the best suggestion from the list generated by the diary-agent.

From Figures 5.8 and 5.9, we selected two preferences to explain in detail (Figure 5.10). They show that QDT-preferences can be translated to OutOfBrain quite easily. The left one expresses the todo-agent's preference for selecting important todo-items as urgent. The right one represents the user-agent's preference for alternating fun activities with boring ones. There are actually two preference needed for that: one for alternation with the prior do-item and one for alternation with the next do-item.

Both preferences are of the type 'count', which means that the agent tries to choose an option[1] that satisfies the preference as often as possible. In other words, in case of the left preference, the most urgent todo-item should ideally have a higher importance value than all other todo-items. In case of the preference on the right-hand side, the agent wants a preferred suggestion that comes with a prior-delta-fun higher than that of most other suggestions. This concludes our elucidation of the preferences used in the adaptive diary assistant.

[1] A todo-item or a suggestion, depending on which agent we are talking about.

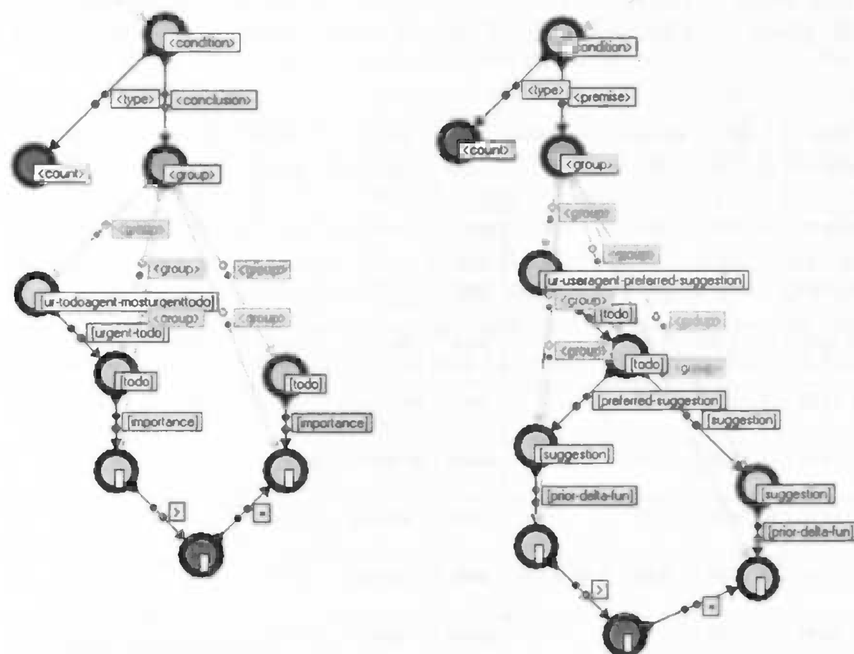


Figure 5.10: The todo-agent's preference for choosing important todo-items as most urgent (left) and the user-agent's preference for alternating fun activities with boring ones (right).

5.5 Test report

To test the diary assistant properly, a large empirical study would be required. One where a group of subjects would have to use the system for several months and then give structured feedback on their findings. Such a test is of course beyond the scope of the project, but we need to conduct some form of testing to prove that the diary assistant is useful. The current Section provides the report of a small test in which the diary assistant performs some typical planning operations. The entire test was performed on Sunday the ninth of July.

The test consists of fourteen do-items (filling one week of the diary, starting at the tenth of July) as well as seven todo-items. This set of items and the planning operations performed on them, serve as a snapshot of the diary assistant in use. Unfortunately, the long-term effects of user-modelling cannot be tested this way. The do-items and todo-items can be found in Table 5.1 and 5.2 respectively. For a screenshot of the actual application during this test, we refer back to Figure 4.8. Times and dates (in the columns from, till and deadline) have been abbreviated. Monday the tenth of July, 11 a.m. for example, becomes "10, 11.00". The attribute values in the four rightmost columns can range from one to ten.

Choosing a todo-item
After putting these twenty-one items on the diary and the todo-list, the user-model needs to be updated. Also, whenever a new item is added, it inherits the average attribute values of all items with the same type and subtype. This is possible thanks to the type-tree (see Figures B.7 and C.5 of the Appendices). The user can still change the values manually of course.

The first step in the test is having the todo-agent select the most urgent todo-item (see Section 5.4). He chooses item number one from Table 5.2. This is an intelligent decision, because the item's deadline is close. Furthermore, it is very important and a lot of fun too.

Also, it does not take much physical effort. The item's only drawback is that it takes a lot of mental effort. Still, given the characteristics of the other six items, it is probably the best choice.

#	*action*	*from*	*till*	*duration*	*deadline*	*type*	*subtype*	*location*	*imprtnce*	*ment eff*	*phys eff*	*fun*
1	email/phonecalls	10, 9.00	10, 11.00	2.00	-	work	correspond ence	IDEA	7	8	1	3
2	staff meeting	10, 11.00	10, 13.00	2.00	-	work	meeting	IDEA	8	9	1	4
3	write project proposal	10, 14.00	10, 18.00	4.00	28, 17.00	work	writing	IDEA	9	10	1	7
4	give math lecture	11, 9.00	11, 11.00	2.00	-	work	teaching	IWI	10	9	2	8
5	email/phonecalls	11, 11.00	11, 11.30	0.30	-	work	correspond ence	IDEA	4	7	1	3
6	attend colleague's presentation	11, 14.00	11, 15.00	1.00	-	leisure	performnce	Botering estr	2	6	1	9
7	project meeting	11, 15.30	11, 16.45	1.15	-	work	meeting	IWI	7	9	2	8
8	write project proposal	12, 9.00	12, 17.00	8.00	28, 17.00	work	writing	IDEA	9	10	1	7
9	soccer training	12, 20.00	12, 22.00	2.00	-	leisure	sports	OZW	2	1	9	8
10	homework correction	13, 9.00	13, 11.00	2.00	17, 11.00	work	correction	home	8	4	2	5
11	email/phonecalls	13, 11.00	13, 11.30	0.30	-	work	correspond ence	home	4	7	1	3
12	clean kitchen	13, 11.30	13, 13.00	1.30	-	househ old	cleaning	home	2	1	7	2
13	coffee with Pete	13, 15.00	13, 16.00	1.00	-	leisure	social	home	2	1	1	9
14	give logic lecture	14, 13.00	14, 15.00	2.00	-	work	teaching	IDEA	10	9	2	8

Table 5.1: The do-items that are used to test the diary assistant. They are supposed to represent a typical week in somebody's diary.

#	*action*	*from*	*till*	*duration*	*deadline*	*type*	*subtype*	*location*	*imprtnce*	*ment eff*	*phys eff*	*fun*
1	prepare logic lecture	-	-	1.00	14, 13.00	work	prepare lecture	IDEA	8	7	1	7
2	prepare math lecture	-	-	1.30	18, 9.00	work	prepare lecture	IDEA	8	7	1	7
3	return library books	-	-	0.10	16, 17.00	work	misc	library	7	1	4	6
4	meeting with student	-	-	1.00	-	work	meeting	IDEA	8	8	2	8
5	homework correction	-	-	1.00	18, 11.00	work	correction	IDEA	8	4	2	5
6	clean bathroom	-	-	1.00	-	househ old	cleaning	home	5	1	7	2
7	pruning	-	-	0.45	-	househ old	garden	home	3	2	7	7

Table 5.2: The todo-items from the test. These are used to test the diary assistant's planning skills.

Generating suggestions
So we agree with the todo-agent that "prepare logic lecture" is the most urgent item on the todo-list. Next, we can have the diary-agent generate suggestions for the todo-item. The list he comes up with is also well informed. It consists of five suggestions instead of the usual ten. This is because the deadline of "prepare logic lecture" is passed after generation of the second suggestion. In such cases, the diary-agent should continue until five suggestions are collected. Obviously, the system should avoid doing an activity after its deadline. So, hopefully, the user-agent will select one of the two suggestion that are before the deadline. The generated suggestions are listed below. Recall that the diary-agent jumps half the todo-item's duration after generating a successful suggestion. All five of them are valid suggestions that do not result in overlap with the items in Table 5.1. However, not all of them are intelligent decisions. It is up to the user-agent and his preferences to select the best suggestion from the list.

suggestions for todo-item #1:
- July 11, 11.30
- July 11, 12.00
- July 13, 13.20
- July 14, 9.00
- July 14, 9.30

Choosing a suggestion
When the list of suggestions for todo-item number one is done, the diary-agent sends it to the user-agent who has to select the best suggestion according to his preferences (see Section 5.4). The choice he comes up with is July 11th, 11.30. We agree with this choice, because it is as early as possible and it introduces a lot of variation (both in subtype and attributes). Furthermore, travel time is minimal. If the human user accepts the team's choice, the todo-item is moved to the diary, on the eleventh of July, at 11.30.

Planning two more todo-items
When we invoke another planning-cycle, the todo-agent chooses todo-item number three: "return library books". This is also a suitable choice, because the item's deadline is close and its effort is low. Mental and physical effort could be interpreted in two ways. First, it could refer to the amount of effort for the entire item, which means that these attributes implicitly hold information about the item's duration. Second, they could refer to the amount of effort per time unit, which necessitates a separate preference for short durations. We chose to go with the first interpretation of effort[1]. Consequently, since returning one's library books is only a matter of minutes, the item has low effort values. According to Table 5.2, returning one's library books is fairly important and fun, which also has a positive effect on the item's urgency.

The corresponding list of suggestions, created by the diary-agent, is given below. We encourage the reader to verify its validity in combination with Table 5.1. The best suggestion according to the user-agent is marked with an asterisk. This result shows that the utilized preferences do not always lead to intelligent behaviour. The chosen suggestion involves a considerable amount of travel time and spare time to kill. Furthermore, the chosen step size of half a duration is too small if the todo-item takes only ten minutes. Fortunately, the user can plan items manually whenever the team of agents makes a poor choice.

---

1 We found that the second interpretation of effort, combined with a preference for short duration makes the todo-agent focus too hard on easy items (short and effortless) and thus he postpones the really urgent tasks too much.

suggestions for todo-item #3:

- July 10, 13.20
- July 10, 13.25
- July 10, 13.30
- July 11, 11.50*
- July 11, 11.55
- July 11, 12.00
- July 11, 12.05
- July 11, 12.10
- July 11, 12.15
- July 11, 12.20

One can also select a todo-item manually. As an example, we make the team of agents plan todo-item number six: "clean bathroom". The list of suggestions for todo-item six is reasonably good. Since the idea should be clear by now, we refrain from providing the list. As a final result, we want to point out that the best suggestion, according to the user-agent, is July 13th, 13.00. This is an intelligent choice, because the suggestion is as early as possible, taking into account the other preferences. Furthermore, the choice involves no travelling whatsoever. Variation in subtype and attributes with the prior do-item (Table 5.1, #12) is poor, but the variation with the next do-item (Table 5.1, #13) makes up for that. This example shows quite clearly that partial matching occurs and that it results in intelligent behaviour.

This concludes the test report and Chapter 5 as well. The sixth and final Chapter of the thesis contains our evaluation, conclusions and future work.

6 Discussion and conclusions

The final Chapter of this thesis contains our final conclusions (Section 6.2) as well as an extensive discussion concerning evaluation and future work (Sections 6.1 and 6.3 respectively). Since this is not an empirical research project, we do not include a separate Chapter for results. Neither do we try to confirm or refute a specific hypothesis. Instead, we evaluate our work with respect to the goals presented in Section 1.2.

6.1 Evaluation

In this Section, we first evaluate our goals (6.1.1). Then we take a look at what was accomplished with respect to OutOfBrain as a programming environment as well as with respect to the design and implementation of the adaptive diary assistant (6.1.2). Finally, the design reconsiderations mentioned in Section 5.1 are further explained in 6.1.3.

6.1.1 Goal evaluation

Before we evaluate the entire array of goals defined in Section 1.2, we want to treat the two main research questions that were raised there, in the context of our two main goals. First, we asked ourselves the question how the QDT architecture could be extended to allow for temporal reasoning. We found an answer in the BDI architecture. After studying both the QDT and the BDI architecture, we came to the conclusion that they could be combined to make a powerful system that utilizes three modalities: one for preferences, one for normalities and one for time. This conclusion is supported by [Dastani et al. 2003] which, however, does not provide any ideas on how to combine QDT and BDI. We, on the other hand, have actually attempted to create such a combination. In designing it, we were reasonably successful.

The second research question we raised in Section 1.2 is whether or not the extended QDT architecture can be implemented in OutOfBrain. Unfortunately, we cannot yet answer this question, since QDT+ was not implemented in OutOfBrain. We did succeed in implementing the standard QDT architecture, but the functionality for temporal reasoning did not make it past the design stage.

The array of goals of this project can be divided into the two main ones and a collection of secondary ones. All of them are defined in Section 1.2. The first main goal is concerned with analysing the issues a developer has to face while creating a multiagent system. We claim that the project is a success with respect to this goal. The project was completed in under a year by a small team of only two developers. However, we went through all the typical stages of MAS

development. Consequently, we encountered many of the typical issues a MAS developer has to face. However small, the project could possibly serve as an example for other MAS developers. In principle, this thesis could guide the development of other multiagent systems. It could also serve others in avoiding known pitfalls. Since MAS is a relatively young field of research, we feel that the need for reference material is stronger than in other disciplines.

Our second main goal is to explore and extend the capabilities of OutOfBrain. We claim that this goal was also pursued successfully. During the project, many facets of OutOfBrain were analysed quite thoroughly. We now have a clear picture of the programming environment's capabilities and this thesis could serve as reference material for people working with OutOfBrain in the future. As for extending the system; when we compare the OutOfBrain that was, at the start of this project, to the current version, we see that a great deal of functionality has been added. Bugs were fixed, usability was improved and built-in methods were created for arithmetic, deletion and duplication of structures[1] and QDT logic of course. Section 6.1.2 further explicates these improvements to OutOfBrain.

The project's secondary goals are concerned with user modelling, agent communication and interface design. With respect to these goals we were not so successful. Especially the user modelling components of the diary assistant do not live up to expectations. This deficit was caused mainly by a lack of time and theoretical background. We return to this topic in Sections 6.1.3 and 6.3.1. As for agent communication, we did find the required time and theoretical background. Consequently, both the design and implementation of communication within the team of agents were worked out properly. The system we ended up with is not that complex with respect to communication, but it does show the potential power of the FIPA standard for agent communication. Finally, let us discuss the secondary goal concerned with interface design. We anticipated beforehand that we would not have the time or theoretical background to design a perfect interface. After all, this is not a study in cognitive ergonomics. Our interface cannot be labelled as especially user-friendly, but it is usable to say the least.

We can summarize the evaluation of our goals as follows. We wanted our project to contain all the facets of a typical MAS development. However, most researchers focus on a small part of a larger system. When our work is compared to other projects, it becomes apparent that our setup was indeed too broad and diverse, especially in relation to the number of man-hours. The goal to analyse the entire developmental process of a MAS can simply not be unified with a good research setup. Due to the broad definition of our goals, we did indeed fail to reach some of them. Still, we are quite content with the results concerning our main goals: analysis of issues in MAS development (especially with respect to QDT) and exploration and extension of OutOfBrain.

6.1.2 Results

Before we move on to the conclusions of the project, we want to evaluate our work with respect to some more concrete topics as opposed to the much broader goals that were evaluated above. In particular, we would like to shed some light on what was and was not achieved during design and implementation. In this context, we evaluate our work in the development of OutOfBrain as well as the adaptive diary assistant created in OutOfBrain.

[1] One can now define data-members as being of a specific structure such as a tree or a linked list. These structure definitions are necessary for easy deletion and duplication of these data-members. We return to this topic below.

OutOfBrain
The most important result from Rockingstone's point of view is that we created the first ever OutOfBrain application. From this case study many new insights were gained. These will aid the further development of OutOfBrain and future applications built with it. The OutOfBrain graph containing the hostgraph and the three agents could serve as an example for future OutOfBrain developers. This is an important achievement since our graph is the only complex piece of OutOfBrain code that exists today. When combined with the overview in Section 5.2 and Appendix B as well as the manual in Appendix A, our source code could be a welcome companion for new developers.

The largest improvement to OutOfBrain is of course that one can now build QDT-agents in OutOfBrain. We will not get into this here. After all, the larger part of this thesis is about this topic. Besides improving OutOfBrain with respect to agents, we made a start with mathematics. The engine always looks for outgoing edges that are labelled with arithmetic operators. Consequently, we can do arithmetic checks as well as arithmetic assignments (Sections 5.11 and 5.12 of Appendix A).

The third large improvement is concerned with data-structures. It is crucial that entire structures can be deleted and duplicated without knowing the exact nodes and edges in them. To do this, one has to be able to define these structures (such as lists and trees or more constrained ones like binary trees). Once the structure is defined, we can use it to duplicate or delete data of that form. In an implication, we only have to provide the structure's root along with the structure's name (e.g. linked list). An edge labelled 'duplicate' or 'delete' tells the engine what to do. This improvement to OutOfBrain is not yet available in the manual provided in Appendix A.

The fourth and final large improvement is actually a collection of smaller improvements, concerned with user friendliness. The main usability improvements that took place during the past year are the extended use of different colours, improved keyboard controls and the introduction of zooming functions. It is safe to say that, although a lot remains to be done, OutOfBrain has already surpassed the usability levels of most textual programming environments due to the use of graphical representations. It is probably especially useful for developers that do not have much background in computer science. We will not get into this claim, because proving it would require another project of the same scale as this one.

Diary Assistant
At the start of this project, there was an older version of OutOfBrain available. All we had to do was enhance it appropriately. As for the diary assistant, we had to start from scratch. However, we did have existing electronic diaries to use as inspiration. Section 5.6 reports on a small test of the application. The diary assistant behaves largely as expected, making reasonably intuitive and intelligent decisions. The intelligent parts of our application can be seen as new results, since no existing diaries that we know of use intelligent agents. This is probably because humans do not trust computers to plan their appointments. Humans rely heavily on intuition during planning. Keeping one's diary is a configuration problem and it is a well known fact that humans solve these better than computers do. However, Boutilier's QDT logic and other cognitively inspired architectures are very intuitive, since they use notions such as desires and preferences. We expect that when humans are asked why they plan a certain item in a certain way, they would report reasons that show great resemblance to the ones utilized by our diary assistant.

Our choice to use more than one agent for a single diary is a poor one from an engineering perspective. Therefore, the development of the diary assistant stops at the end of this project, although it is far from done. Before we move on to the next Section, we would

like to very briefly present the results with respect to the application. First of all, the application is quite slow. One has to wait up to two minutes when the team of agents is busy reasoning and communicating, which makes it less convenient in use. There are several possible causes to this problem. The ones below are presumably all partly responsible for the lack of speed.

First of all, the problem's fundamental complexity is exponential. Humans guide their decisions with intuition, dramatically pruning the search-tree, so they do not suffer from the complexity of the problem. Second, it could be that OutOfBrain itself is slow. High level programming languages are known to be a bit slow sometimes (Python for example). OutOfBrain is still under development and we know for a fact that the search algorithm can still be optimized in many ways. We know that sub-graph isomorphism is NP-complete [Andries et al. 1999]. But due to the use of a unique index for each node, complexity approximates $O(n * log(n))$ which is fast for graph rewriting[1].

We rule out the possibility that Delphi causes the problem. It is not as fast as C++, but it is a programming language with a long history (one of the longest) and many great developers. The third cause could be that the communication between the Delphi parts (view and control) and the OutOfBrain part (model) slows things down. The speed of this type of communication was already improved during the project, but maybe it needs some more work.

It could also be the case that our implementation of the diary assistant in OutOfBrain is suboptimal. Most certainly, the communication between the different agents slows the system down substantially. Finally, the system probably performs superfluous checks and some subroutines might be solved more efficiently altogether. This concludes the results Section. We now discuss the design reconsiderations a bit further and after that, it is time to present our conclusions.

6.1.3 Reasons for design reconsideration

Many components of the design were not implemented. These were already mentioned in Section 5.1. Below, we briefly discuss the reasons for these design reconsiderations. They are caused by three separate problems. First of all, we used an incremental design strategy. The design changed in many ways during the implementation phase. Second, we had only one year and two part-time developers to complete the application. The first half of this period was used mainly for literature study and brainstorm sessions, which left only half a year for design and implementation. Our third problem is that we lack the theoretical background on the topic of learning in a qualitative MAS architecture. There is probably more than enough good literature on BDI-learning, but since QDT is less popular, we did not find appropriate articles on learning. The architecture we used for our design is based on QDT, but we added functionality for time and events from BDI. This makes good literature even harder to come by. Then again, if such literature did exist, our design would not have been a novelty.

The BDI components that we were supposed to add to the QDT architecture did not take solid form. This is most regrettable, since a general purpose MAS architecture must be able to deal with time and events. An agent should be able to guide his decisions by the history of percepts as opposed to only the last observation. According to [Russell and Norvig 1995] as well as [Wooldridge 2002], all but the simplest agent types possess a history of percepts. Furthermore, if general purpose learning is the goal, the analysis of past states and events is absolutely indispensable (in most domains). For these reasons, we believe that the

[1] With respect to worst case time complexity, sub-graph isomorphism is still NP-complete. It is the average time complexity that is now tractable, thanks to the unique indexing approach.

choice to add time and events to QDT is a good one. This is affirmed by Mehdi Dastani and his colleagues. They claim that QDT and BDI can (and should) be combined in several ways, without fear of incompatibility [Dastani et al. 2003].

Now, why is it that QDT(+) is not used extensively in our diary assistant? The system uses around twenty preferences, but no normalities or time whatsoever. Apparently, it is harder to define these things than we supposed. It proved to be even harder to distill preferences and normalities from a text-interface. The intuitive nature of QDT(+) makes the gap between raw input and workable formulas a great deal smaller. Nevertheless, we underestimated this part of the problem. The importance of the modality time for a general purpose MAS architecture was already discussed above. The necessity of normalities is comparably great. In combination with preferences, it allows agents to behave rationally using a qualitative version of Savage's expected utility maximization. We do believe however, that the use of normalities can legitimately be avoided in the domain of diary-keeping.

6.2 Conclusions

This project follows a protocol that is theoretical in nature. We do not perform any empirical research and we have no hypotheses to test. Consequently, the conclusions in this Section are based on other, less exact measures. In short, the two meta-goals of this project are for me to learn about MAS development and for Rockingstone to get some hands-on experience with OutOfBrain. We argue below that both these meta-goals were reached. From the goal evaluation in 6.1.1, we arrive at the following conclusions.

Issues in MAS development
The question we raised in the context of MAS development issues is: "How can the QDT architecture be extended to allow for temporal reasoning?". The answer is that it can be done by combining QDT with the temporal component of BDI. We did not prove that it can actually be implemented in practice, but neither did we encounter a reason to assume the contrary.

As for the context of this question, we found that it is hard to design and implement a new architecture that is general purpose and fit for a specific application at the same time. Luckily, there are many existing implementations of MAS architectures to choose from, the BDI architecture in particular has been implemented successfully several times. Of course, we did not actually create a new architecture. It was more a matter of putting two existing ideas together. It would be an impossible task to list all the issues one has to face during MAS development. But many of the ones we encountered have been reported upon in this thesis. Before we move to our conclusions with respect to OutOfBrain, we put forward some conclusions concerning the design of MAS architectures.

One of the more interesting issues we encountered is that it can be hard to maintain qualitativity. The diary assistant is less qualitative than we would like it to be. This is in part due to the fact that QDT is fundamentally less qualitative than BDI. However, it is also caused by the translation of possible world orderings to integer scales and their multiplication to obtain a measure of expected utility. Furthermore, the domain specific information is fundamentally quantitative, since the agents have to compare times, dates and integer attributes (importance, mental effort, physical effort and fun). Despite these compromises, we conclude that our approach is better than Savage's classical method in terms of complexity and intuitiveness.

Another interesting issue arises in the combination of preferences and normalities into a measure of expected utility. As stated before, we believe that the way we combine them is equivalent to the proposed method in [Boutilier 1994] (compare Sections 4.3.2 and 4.3.3 to Definition 2.18). QDT is nothing more than two CO models combined. There are only two differences between preferences and normalities. The first is simply their interpretation. The second is that they are treated differently when combined in Definition 2.18. First, the default closure of the knowledge base is determined. Then, this closure is considered to be true knowledge and along with the preferences, action set dominance is calculated. We adopted this priority of normalities over preferences in our OutOfBrain implementation of QDT.

An issue that should also be mentioned here is that we had trouble finding a good task distribution. We found that which agent does what is far from trivial and we switched strategies several times before finding a workable task distribution. The agents have access and control over part of the environment and communication is both costly and complicated. Therefore one should minimize the amount of communication through careful consideration over who does what. [Dignum et al. 2001] discusses the topic of dynamic task allocation. This is an interesting topic, but it is beyond the scope of this project, since we have a fixed team where every agent has a fixed set of tasks.

Another interesting problem we encountered, the last one we put forward here, is determining the amount of parallelism in the system. The absolute minimum here is three parallel processes, one for each agent. Initially, each agent consisted of several processes, but we found that this complicates things unnecessarily. The advantage of having only one process per agent is that it is clear which state an agent is in. Consequently, the dynamics of the system are easier to comprehend. Furthermore, we get an environment that is less dynamic and hence easier to handle. If there is parallelism in each agent, the environment cannot only change unexpectedly through actions of others, but also through actions in other processes within the same agent. This is a problem that has to be reckoned with. The advantage of simulated parallelism within agents lies in the time advantage. An agent can keep working while waiting for a reply. The speed of the entire team would not suffer much from one agent's slow reply. Needless to say, true parallelism makes the time advantage even greater. We now move on to our conclusions with respect to OutOfBrain.

OutOfBrain

Next we devote some attention to our conclusions on exploring and extending the capabilities of the OutOfBrain programming environment. The question that was raised in this context is: "Can the QDT architecture, extended with a temporal modality, be integrated into OutOfBrain?" Unfortunately, we cannot provide an answer. We can say however that QDT as it was designed by Craig Boutilier was successfully implemented and is now an integral part of the OutOfBrain programming environment.

As far as exploring the capabilities of OutOfBrain is concerned, we had an entire island of unexplored territory to digest[1]. The environment turns out to be what it claims: A versatile graphical programming environment for artificial intelligence programming. It has some particularly strong points that we would like to emphasize a bit.

First of all, it is easy to learn. Anyone with some programming experience can get used to it in a couple of weeks. Creating software is no longer an activity for a small elite. As more and more people start doing it, there is a strong demand for easily accessible programming languages. We feel that OutOfBrain could be such a language and it might also serve educational purposes some day, because of its intuitiveness. The graphical

[1] Tim Samshuijzen was of course very well acquainted with OutOfBrain at the start of this project. Without his guidance we would not have come very far.

representations probably provide the creative process with an extra dimension for people that are visually oriented (as opposed to auditory oriented). The type of graphs used in OutOfBrain and other graph rewriting mechanisms prove to be a universal data-structure that is easy to comprehend. Furthermore, we think that OutOfBrain is relatively user friendly compared to textual languages. This holds especially for developers without programming experience. Many of the available packages for agent programming (at a higher level of abstraction) also use fairly complex representations. However, there are exceptions such as the highly comprehensible environment 3APL [Hindriks et al. 1999].

We expect that the most promising future for OutOfBrain lies in hybrid systems like our diary assistant. OutOfBrain simply lacks the expressive power to handle all the components of a typical piece of software. In such hybrid settings, OutOfBrain should be used to implement internal representations and reasoning (model), while a textual language takes care of input and output. Reasoning in OutOfBrain is well suited for abstract representations, but a language like C++, Java or Delphi should be used to convert the raw data coming from the environment (real or virtual) to logical formulas. A textual language should also be used to handle view and control of the application, so that existing libraries can be used.

Another strong point of OutOfBrain is that it allows for distributed development and run-time programming. The system uses client/server communication for several reasons. First of all, it is used for communication between view and model and between model and control. Any programming language can be in a hybrid setting with OutOfBrain thanks to the TCP/IP communication standard. This can either take place on one and the same computer (different ports on 127.0.0.1) or on different computers or robots connected via internet. Second, TCP/IP is used for communication between the developer tool and the running application during run-time programming, allowing for a convenient way of debugging. Finally, the client/server setup allows for several people to connect their developer tools to the same application, thus leading to distributed development.

Despite the versatility and power of OutOfBrain, there was a lot of room for improvement at the start of this project. During the past year, OutOfBrain has been extended in many ways. Bugs were fixed, usability was improved and built-in methods were created for QDT logic, arithmetic and deletion and duplication of structures. Of course, a lot of work still remains to be done (Section 6.3.2).

6.3 Future work

Driven by our diverse array of goals and motivations, we have come across many new insights. Most of them are only new to ourselves, but some of them could be useful to others as well. Because of the broad scope of the project, we touched upon many sidetracks, not nearly exploring them all. Consequently, we have many ideas for future work. First, we present a general view of expected future work related to our endeavour. Then, we discuss some more concrete potential future work with respect to the different branches of our project.

First and foremost, we expect that a QDT+ like architecture should and will be properly designed and implemented. BDI is currently the leading MAS architecture and most researchers will probably stick to that. However, a change of food whets the appetite and it could be risky to set our hopes on only one approach. BDI is a general purpose architecture, but it could turn out that an architecture based on QDT will perform better in certain domains. This claim is mainly backed by the fact that BDI is inseparable from binary mental attitudes which leads to non-deterministic choices. From a cognitive standpoint, this is a good and natural property. However, from an engineering point of view, it is sometimes desirable to

resolve such conflicts in an informed way. QDT is more capable of doing so. More on future work with respect to QDT+ in Section 6.3.1.

Second, the OutOfBrain programming environment is prone to spread its wings and hopefully it will enjoy wide support. The environment 3APL, developed by Dastani and colleagues [Hindriks et al. 1999], should serve as an example to Rockingstone in the years to come. The development of yet another MAS programming environment might seem futile at first, but OutOfBrain has different capabilities. The environment 3APL is entirely devoted to implementing reasoning agents. OutOfBrain is more suitable for implementing other components of an application as well (arithmetic and complex data-manipulation for example). Section 6.3.2 contains some concrete ideas for improving OutOfBrain.

Our endeavours toward developing an adaptive diary assistant were not so fruitful. Our approach is poor from an engineering perspective, because a single diary assistant is not a true MAS; it is not distributed in space. Therefore, it should not be run by a team of agents. Consequently, this particular diary assistant will not enjoy any future work. Concerning future work, we feel that it is inevitable that MAS and interconnected PDA diaries will form a healthy marriage in the near future. See Section 6.3.3 for some more thoughts on that.

6.3.1 Improvements to QDT+

During our discussion of future work, we focus on QDT+, because scientifically speaking, it is the most interesting branch of the project. We stand by the claim that QDT is a good starting point for a general purpose MAS architecture. The negative evaluation of our implementation in Section 6.1 does not mean that this hope should be abandoned. It only means that a lot of work remains to be done. Other researchers should see the potential and join the QDT movement. So far, its big brother BDI has drawn away most of the attention.

But how does one go about creating an architecture like QDT+? In particular, how can we develop an architecture that utilizes preferences and normalities as well as time? Using BDI's temporal modality seems to be a good move. Our own findings, combined with [Dastani et al. 2003] lead to this conclusion. Obviously, the history of the environment and the agents needs to be stored in order to achieve such a tri-modal system. We have no doubt that this can be done. In fact, Section 4.2.1 suggests that it is not so hard to achieve. Once we have such a storing mechanism, combined with the two other modalities, preference and normality[1], the question rises of how to derive preferences and normalities (possibly containing temporal clauses) from these histories of percepts.

We had hoped to answer this question, but we can only guess at it, due in part to the shortage of time and literature on QDT-like learning. It proved to be very difficult to extract useful knowledge from histories of percepts. The intuitive nature of the architecture partly bridges the gap between inputs and new knowledge. In particular, it allows user and computer to understand each other, because preferences, normalities and time are meaningful to both. This is nice, but the question of how to gain new knowledge from histories of percepts remains. The answer depends partly on the mode of input. Are we using natural language, a formal syntax or perhaps just buttons and single-word input-fields? The first step toward the mapping of percept histories to new knowledge is to turn raw input into meaningful logical formulas. To do this in a general purpose way turned out to be too much to hope for in the scope of this project. We used ad hoc, domain specific solutions instead.

[1] Recall that in our design, time is a tree of possible worlds that branches into the future and is linear in the past. Each possible world in this tree has two more modalities, so there is an extremely wide variety of sentences in the language. It ranges from first order statements about the environment to sentences that combine temporal operators with mental attitudes.

Once this obstacle is out of the way (rendering the input suitable for OutOfBrain or another agent programming language) there are at least two ways to proceed. One possibility is to allow agents to add new preferences and normalities to their knowledge base. It is the best solution, but also the hardest one to achieve. It would allow an agent to learn how to behave in novel situations but it would require tremendous intelligence. Agents would have to be able to learn the meaning of new symbols, which is problematic, to say the least. A different approach is to make sure that all possible preferences and normalities are already present after initialization, so every rule an agent could possibly want to utilize is there from the beginning. However, only the ones in some initial subset and the ones that follow from experience are activated. The rules can be turned on and off, but they could also be outfitted with dynamic weights that determine how strong the preference or normality is. This latter option compromises the qualitativity of the system and the former variation is therefore preferred in most cases. This learning strategy would only work in domains where all possible preferences and normalities can be determined during design time. In many settings, this requirement is not met, not in the real world and not in many experimental settings and virtual worlds either. After all, MAS have to be geared to operate in dynamic, uncertain environments.

6.3.2 Improvements to OutOfBrain

Rockingstone aims to create a full-fledged graphical programming environment for intelligent agents. The release date of OutOfBrain 1.0 has not yet been announced, since a lot of work remains to be done. This Section explains how OutOfBrain will be improved in the years to come.

A piece of future work directly related to this project is to create a storing mechanism for the history of an agent's percepts and inner states. As stated before, storing histories is an absolute necessity for intelligent agents. Thanks to the current project, the development of such a storing mechanism for OutOfBrain is closer than ever. First of all, we provide an abstract design for it in Section 4.2. Second, we determined and discussed some issues concerning temporal reasoning.

Another topic for future research and development is OutOfBrain's search engine. Due to the hacks discussed in Section 3.1.7, the current engine is considerably faster than that of general graph rewriting. However, a lot of work remains to be done to optimize the engine. Since OutOfBrain's complexity is on the edge of tractability, this topic should enjoy a lot of attention from future OutOfBrain developers.

The third opportunity for future work we would like to put forward is improving the client/server communication (Appendix A, Chapter 7). In particular, it should be speeded up. TCP/IP communication forms an integral part of the OutOfBrain environment. It is used for communication between the different parts of an application (The Delphi components and the OutOfBrain components in our case). We think that this problem is partly responsible for our adaptive diary assistant's lack of speed (see Section 6.1.2). In the future, TCP/IP will also be used for distributed development. Developers from different countries will then be able to team up and work on the same OutOfBrain program simultaneously. The client/server component also has a promising future in true MAS settings.

There are two more improvements to OutOfBrain we would like to mention here. Once the first retail version of OutOfBrain is released, people should start working on libraries containing predefined functions, both on the Delphi level (under water) and the OutOfBrain level. This would allow OutOfBrain developers to make use of existing view components, control components, complex data-manipulation methods and hardware access

methods. Finally, the usability of the OutOfBrain toolbox is apt to be improved with sophisticated editing and viewing capabilities. It should be clear that there is much more to say about improvements to OutOfBrain. However, we choose to leave at this for now.

6.3.3 A network of personal digital assistants (PDA's)

Now that future work with respect to OutOfBrain and QDT+ is covered, we would like to shed some light on our expectations concerning electronic diaries. As stated before, a diary should be run by a single agent that can use all the available data instead of by a team of agents with partial access and control. The MAS approach should only be used in true MAS situations (distributedness, differing goals and cooperation or competition). A single diary is clearly not such a situation. We only took the MAS approach, because it is interesting from a scientific point of view.

Now consider the situation where there is an entire network of adaptive diary assistants. Each diary is controlled by a single agent, but since there are many geographically distributed diaries, we find ourselves in a true MAS setting. The requirement of distributedness is met, because every agent is situated on a physically separate PDA. There are differing goals because each agent represents a different person. Finally, we can have both cooperation and competition in this setting. Agents can work together by bringing their masters together in teams. But they can also negotiate about making appointments. In such a setting, the personal agents have to deal with team formation and dynamic task allocation [Dignum et al. 2001].

Imagine a company where everybody has such a personal assistant. The company's leaders are having a meeting in the boardroom. Since they are all extremely busy, it is hard to come up with a suitable date and time for the next meeting. This is when the chairman grabs his or her PDA and presses a single button. The agents wake up and start negotiating. The next meeting has to be compatible with as many of the board members' appointments and preferences as possible. The agents also communicate with the digital assistants of the colleagues that are not in the room. The different divisions of the company could act as teams, the members always considering each other's preferences. Not everyone will be satisfied by the solution of the agent network. But hopefully, the new appointment will be close to optimal for as many people as possible. We claim that such a multiagent system will soon move from the realm of science fiction to reality.

The adaptive diary assistant created during this project can be improved in many ways, but since we decided not to continue its development, we leave those to the reader's imagination. This was the final Chapter of the thesis. All that remains are the references and the appendices.

References

Aarts, E., Marzano, S. (2003) *The New Everyday: Views on Ambient Intelligence.* 010 Publishers, Rotterdam, The Netherlands

Andries, M., Engels, G., Habel, A., Hoffmann, B., Kreowski, H. J., Kuske, S., Plump, D., Schürr, A., Taentzer, G. (1999) Graph transformation for specification and programming. *Science of Computer Programming, vol. 4, no. 1, pp. 1-54*

Baader, F., Horrocks, I., Sattler, U. (2005) Description logics as ontology languages for the semantic web. In: *Hutter, D., Stephan, W. (eds.): Mechanizing Mathematical Reasoning: Essays in Honor of Jörg Siekmann on the Occasion of His 60th Birthday (number 2605 in Lecture Notes in Artificial Intelligence), pp. 228-248.* Springer-Verlag, Berlin, Germany

Blostein, D., Fahmy, H., Grbavec, A. (1995) Practical use of graph rewriting. *Technical report No. 95-373, Computing and Information Science, Queen's University, Kingston*

Boella, G., Hulstijn, J., Van Der Torre, L. (2005) Decision-theoretic deliberation in resource bounded self-aware agents. In: *McIlraith, S., Peppas, P., Thielscher, M. (eds.): Proceedings of the Seventh International Symposium on Logical Formalizations of Commonsense Reasoning, Corfu, Greece, editors, pp. ?-?.* Dresden University Technical Report

Boutilier, C. (1994) Towards a logic for qualitative decision theory. In: *Doyle, J., Sandewall, E., Torasso, P. (eds.): Proceedings of Principles of Knowledge Representation and Reasoning, pp. 75-86.* Morgan Kaufmann Publishers, San Mateo, California, USA

Bratman, M. E. (1987) *Intention, Plans, and Practical Reason.* Harvard University Press, Cambridge, Massachusetts, USA

Broersen, J., Dastani, M., Hulstijn, J., Van Der Torre, L. (2002) Goal generation in the BOID architecture. *Cognitive Science Quarterly, vol. 2, no. 3-4, pp. 428-447*

Cohen, P. R., Levesque, H. J. (1990) Intention is choice with commitment. *Artificial Intelligence, no. 42, pp. 213-261*

Dastani, M., Hulstijn, J., Van Der Torre, L. (2005) How to decide what to do? *European Journal of Operational Research, vol. 160, no. 3, pp. 762-784*

Dignum, F., Dunin-Kęplicz, B., Verbrugge, R. (2001) Creating collective intention through dialogue. *Logic Journal of the IGPL, vol. 9, no. 1, pp. 289-303*

Doyle, J., Thomason, R. H. (1999) Background to qualitative decision theory. *AI Magazine, vol. 20, no. 2, pp. 55-68*

Dunin-Kęplicz, B., Verbrugge, R. (2002) Collective intentions. *Fundamenta Informaticae, vol. 51, no. 3, pp. 271-295*

Dunin-Kęplicz, B., Verbrugge, R. (2004) A tuning machine for cooperative problem solving. *Fundamenta Informaticae, vol. 63, no. 2-3, pp. 283-307*

Foundation for Intelligent Physical Agents (2001) FIPA ACL message structure specification. *Technical Report XC00061*

Fong, T., Nourbakhsh, I., Dautenhahn, K. (2003) A survey of socially interactive robots. *Robotics and Autonomous Systems, no. 42, pp. 143-166*

Hindriks, K.V., De Boer, F.S., Van Der Hoek, W., Meyer, J.-J. Ch. (1999) Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems, vol. 2, no. 4, pp. 357-401*

Jensen, K. (1992) *Coloured Petri Nets. Basic Concepts, Analysis, Methods and Practical Use. Volume 1: Basic Concepts.* Springer-Verlag, Berlin, Germany

Kahl, W. (2002) A relation-algebraic approach to graph structure transformation. *Technical report No. 2002-03, Computer Science, Bundeswehr University, München*

Lewis, D. (1973) *Counterfactuals.* Oxford University Press, Oxford, UK

Luck, M., McBurney, P., Shehory, O., Willmott, S. (2005) *Agent Technology: Computing as Interaction.* University of Southampton Publishers, Southampton, UK

Nardi, D., Brachman, R. J. (2002) An introduction to description logics. In: *Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D. Patel-Schneider, P. F. (eds.): The Description Logic Handbook, pp. 5-44.* Cambridge University Press, Cambridge, UK

Pearl, J. (1990) System Z: a natural ordering of defaults with tractable applications to default reasoning. In: *Vardi, M. (ed.): Proceedings of Theoretical Aspects of Reasoning about Knowledge, pp. 121-135.* Morgan Kaufmann Publishers, San Mateo, California, USA

Ramsey, F. P. (1926) Truth and probability. In: *Mellor, C. (ed.): Foundations: Essays in Philosophy, Logics, Mathematics and Economics, pp. 58-100.* Routledge and Kegan Paul, London, UK

Rao, A. S., Georgeff, M. P. (1991) Modelling rational agents within a BDI-architecture. In: *Allen, J., Fikes, R., Sandewall, E. (eds.): Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning, pp. 473-484.* Morgan Kaufmann Publishers, San Mateo, California, USA

Rao, A. S., Georgeff, M. P. (1995) BDI agents: from theory to practice. In: *Lesser, V. (ed.): Proceedings of the First International Conference on Multiagent Systems, pp. 312-319.* AAAI Press/MIT Press, San Fransisco, California, USA

Reiter, R. (1980) A logic for default reasoning. *Artificial Intelligence, no. 13, pp. 81-132*

Rich, E. (1999) Users are individuals: individualizing user models. *International Journal on Human-Computer Studies, no. 51, pp. 323-339*

Roediger, H. J. (1990) Implicit memory: retention without remembering. *American Psychologist, vol. 45, no. 9, pp. 1043-1056*

Rumbaugh, J., Booch, G., Jacobson, I. (1998) *Unified Modelling Language Reference Manual.* Addison-Wesley Professional, Boston, Massachusetts, USA

Russell, S., Norvig, P. (1995) *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, New Jersey, USA

Savage, L. (1954) *Foundations of Statistics.* Dover Publications, Mineola, New York, USA

Searle, J. R. (1969) *Speech Acts: An Essay in the Philosophy of Language.* Cambridge University Press, Cambridge, UK

Simon, H. A. (1987) Bounded rationality. In: *Eatwell, J., Millgate, J., Newman, P. (eds.): The New Palgrave: A Dictionary of Economics, pp. ?-?.* Macmillan Publishers, Basingstoke, UK

Stone, P., Veloso, M. (2000) Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots, vol. 8, no. 3, pp. 345-383*

Tan, S.-W., Pearl, J. (1994) Qualitative decision theory. In: *Editors unknown: Proceedings of the Twelfth National Conference on Artificial Intelligence, pp. 928-933.* AAAI Press/MIT Press, San Fransisco, California, USA

Taatgen, N. A., Lee, F. J. (2003) Production compilation: a simple mechanism to model complex skill acquisition. *Human Factors, vol. 45, no. 1, pp. 61-76*

Weyns, D., Holvoet, T. (2004) A coloured Petri net for regional synchronization in situated multi-agent systems. In: *Editors unknown: First International Workshop on Petri Nets and Coordination, pp. ?-?.* PNC, Bologna, Italy

Wooldridge, M. (2002) *An Introduction to Multiagent Systems.* Wiley & Sons Press, Hoboken, New Jersey, USA

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

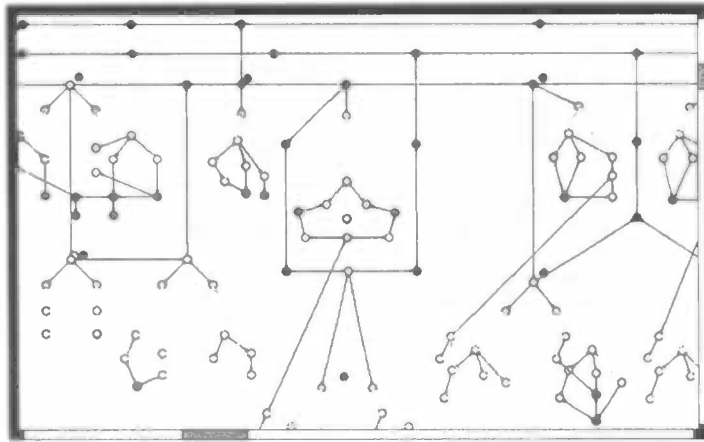
[illegible]

[illegible]

[illegible]

Appendix A: OutOfBrain manual

OutOfBrain



Reference Manual
version 1.01

Rockingstone Robotics B.V.
Wageningen, The Netherlands, 2006

Contents

1 Introduction

2 The OutOfBrain Architecture

3 The OutOfBrain Developer (OOBD)
- 3.1 Opening the OOBD application
- 3.2 Creating a new graph
- 3.2 Opening a graph file
- 3.3 Saving a graph to file
- 3.4 Exiting OOBD
- 3.5 Scrolling though a graph
- 3.6 Zooming out

4 Editing graphs in OutOfBrain Developer
- 4.1 Adding a vertex
- 4.2 Deleting a vertex
- 4.3 Moving a vertex
- 4.4 Labeling a vertex
- 4.5 Selecting and deselecting multiple vertices
- 4.6 Moving multiple vertices
- 4.7 Copying and pasting
- 4.8 Adding an edge
- 4.9 Deleting an edge
- 4.10 Moving an edge
- 4.11 Naming an edge
- 4.12 Navigating across edges

5 The OutOfBrain Virtual Machine (OOBVM)
- 5.1 <group>
- 5.2 <process>
- 5.3 <branch>
- 5.4 <execute>
- 5.5 <debug_pause> and <debug_step>
- 5.6 <implication>
- 5.7 <condition>
- 5.8 <wait>
- 5.9 <candidate> edges
- 5.10 <reference> and <match> edges
- 5.11 Conditional operators
- 5.12 Assignment operators

6 OutOfBrain Agents
- 6.1 The Agent Manager
- 6.2 <voluntaries>
- 6.3 <involuntaries>

6.4 <constraints>
6.5 <consistencies>
6.6 <predictions>
6.7 <preferences>
6.8 <dislikes>
6.9 The single-goal-agent
6.10 The agent types
6.11 An example of an agent

7 TCP communication
7.1 <tcpserver>
7.2 <tcpclient>
7.3 Sending and receiving data

1 Introduction

OutOfBrain is an AI development tool based primarily on the principles of graph rewriting. Unlike most other programming environments, OutOfBrain is not code-oriented. Programming and debugging is done in run-time with OutOfBrain Developer, a graph-oriented graphical user interface which ables the programmer to change the running program without having to shut it down or having to recompile. The client-server architecture allows multiple programmers to work on the same 'source' simultaneously and each programmer can directly follow the work of other programmers. The 'non-code' IDE makes it easy for developers and researchers to communicate their ideas and observations. OutOfBrain is an executable graph combined with the principles of graph-rewriting and autonomous agent technologies. Multiple agents can be created to work together to solve complex problems and achieve goals.

The philosophy behind the use of graphs as opposed to code, is that all data structures can be represented as graphs. In general, code-oriented data structures are limited to records, collections, lists, trees, and combinations of these. Graphs, however, can directly describe all types of structures in a more direct and intuitive way. The developers of OutOfBrain expect the advantages of the use of graphs to become apparent when dealing with complex problems, such as those present in the field of AI.

2 The OutOfBrain architecture

The graph-structures in OutOfBrain are stored in an in-memory graph database. The content of this database is viewable with the OutOfBrain Developer (OOBD). The same content is parsed and processed by the OutOfBrain Virtual Machine (OOBVM), integrated into the OutOfBrain Server. Multiple OOBD clients can connect to the server, allowing multiple programmers to work on the same "source". Interfacing with applications is done via TCP, as TCP servers and clients can be constructed in the form of graphs.

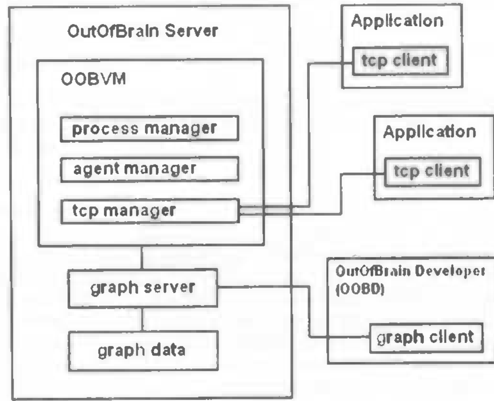


Figure 2.1 *The OutOfBrain architecture*

3 The OutOfBrain Developer (OOBD)

This chapter describes the available commands for opening, saving, creating and navigating graphs in OutOfBrain Developer (OOBD). OOBD is a stand-alone version of OutOfBrain, complete with viewer/editor and integrated OOBVM. There is also a client version called OutOfBrain Developer Client (OOBDC), which can connect to the server version of the OOBVM. In this chapter, however, we just describe the OOBD program.

3.1 Opening the OOBD application

Run the application "outofbrain.exe" to launch OOBD. By default, the program will open the graph file named "main.gd". If no such file exists, an empty graph named "main.gd" is created, but not saved to disk. OOBD is a simple interface with just a single menu called "File". The white background is the graph itself.

3.2 Creating a new graph

To create a new graph, select "New" in the File menu. OOBD will prompt to ask whether the currently opened graph should be saved. Press the appropriate button and a new empty graph will be created called "untitled.gd". At this point the new graph is not yet saved to disk.

3.2 Opening a graph file

To open a graph file, select "Open" in the File menu (or press Ctrl-N). A file-dialog with the caption "Open" will be displayed. Select the desired graph file and press the Open button. Before opening the selected graph file, OOBD will prompt to ask whether the currently opened graph should be saved.

3.3 Saving a graph to file

To save graph to disk, select "Save" in the File menu (or press Ctrl-S). If the graph is titled "untitled.gd", then OOBD will display a "Save as" dialog. To save the graph under a different file name than it currently has, select "Save As..." in the File menu. OOBD will display a "Save as" dialog.

3.4 Exiting OOBD

To exit OOBD, select "Save and exit" in the File menu (or press Ctrl-E). OOBD will save the changes to the graph file and exit without prompting. To exit OOBD without saving changes to the graph file, select "Exit without saving" in the File menu (or press Ctrl-W). Another method to close OOBD is to press the Close in the system menu or by pressing the close-cross button in the upper right corner. This way of closing will save the changes automatically before shutting down.

3.5 Scrolling through a graph

If a graph does not fit within the screen window, OOBD will show scrollbars on the bottom and right sides of the window. These scrollbars will adjust automatically when the lowest vertex or the far-right vertex is moved. To move the window over another part of the graph, click and drag the appropriate scrollbars. The mouse-wheel operates the vertical scrollbar. The mouse-wheel operates the horizontal scrollbar while holding down the Ctrl-key. To move the window over the graph using the keyboard, use the arrow keys. To move the window up or down over the graph, use the Page up and Page down keys, respectively. The Home key positions the window at the top of the graph. To position the window at the top-left-hand corner of the graph, press Ctrl-Home. The End key positions the window at the bottom of the graph.

3.6 Zooming out

Graphs can become quite large, so it is sometimes difficult to navigate to the desired section. In this situation the zoom-function can be helpful. To zoom out, press the Z-key on the keyboard. The OOBD will then show the graph with a zoom-out-factor of 4. At this scale, the labels are not shown and cannot be edited. The border of the original view-window is displayed as a rectangle. To return to the original unzoomed view-window, press the R-key on the keyboard. When moving the mouse over the zoomed-out graph, a similar rectangle is displayed around the mouse cursor. To zoom in onto this rectangle, either press the right mouse button or press the Z-key. To zoom out temporarily, hold down the right mouse button. The view-window rectangle can then be moved to another area, and will become the new view window when releasing the right mouse button. When in zoom-out mode, you can center the view window rectangle by pressing the M-key on the keyboard.

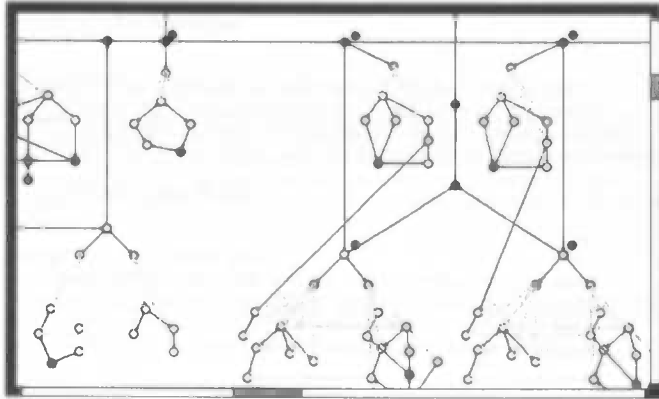


Figure 3.1 *Zooming out.*

4 Editing graphs in OutOfBrain Developer

This chapter describes the available commands for altering graphs. Whenever the text uses the terms "click" or "mouse-button", the left-mouse button is implied. Whenever the text uses the term "key", a button on the keyboard is implied. The chosen terminology for graph components are "vertices" and "edges". In OutOfBrain edges are directional, directed from the "from-vertex" and directed to the "to-vertex". Multiple edges between vertices are supported. Edges with the same from-vertex and to-vertex are also supported. "Dangling edges" are not supported. Vertices and edges each have a one-line textual label. An edge with from-vertex A and to-vertex B is said to be A's "from-egde" and B's "to-edge".

4.1 Adding a vertex

To add a vertex, place the mouse cursor at the position where the vertex is to be placed and press the Insert key. A new vertex will be created with an empty label.

4.2 Deleting a vertex

To delete a vertex, first click the appropriate vertex. Be sure to click the center part of the vertex, and not the black border. The vertex now has a red border showing that it is selected. To delete the selected vertex, click the Delete key. The Delete key deletes all selected vertices. Deleting a vertex will automatically delete all connected edges.

4.3 Moving a vertex

To move a vertex, click and drag the appropriate vertext with the mouse. Be sure to click on the center part of the vertex, and not the black border.

4.4 Labeling a vertex

Each vertex has a label. To assign or change the label-text, click the label with the mouse. A blinking caret will display, allowing text to be entered with the keyboard. Once edited, press the Enter key to save the entered text. Alternatively, click the mouse outside the label to save the text. To cancel the editing, press the Escape key to restore the original label value. When the caret shows, Ctrl-C will copy the text onto the clipboard. Ctrl-C will replace the label-text with that on the clipboard.

4.5 Selecting and deselecting multiple vertices

To select a single vertex, simply click the vertex. Be sure to click on the center part of the vertex, and not the black border. Selecting a vertex in this way will deselect any other selected vertices. To select more vertices, hold the Ctrl key while clicking vertices one-by-one. To deselect a vertex, hold the Ctrl key while clicking the selected vertex. To deselect all vertices, click anywhere in an empty (white) area.
Another method for selecting a group of vertices that are positioned close to each other, is to position the mouse cursor over a white area, hold down the mouse-button, and drag the red box that appears over the group of vertices. Once all desired vertices are selected, release the mouse-button. This also works in zoom-out mode.

4.6 Moving multiple vertices

To move a group of selected vertices, click and drag any one of the selected vertices, drag the group to the desired position and release the mouse-button. This also works in zoom-out mode.

4.7 Copying and pasting

To copy a selected group of vertices, press Ctrl-C. This will place all selected vertices and interconnecting edges onto the clipboard. To paste the group of vertices and edges, scroll to the desired area and press Ctrl-V. Upon doing so, the original group of vertices will become deselected, and the newly placed group will be selected.

4.8 Adding an edge

To connect an edge from one vertex to another, click the black border of the from-vertext. While holding down the mouse-button, drag the edge to the to-vertex. Release the mouse-button and a new vertex with an empty label will be created. It is possible to add multiple edges between two vertices, and to add edges from and to the same vertex.

4.9 Deleting an edge

To delete an edge, first click the appropriate edge on either the arrow head, the arrow tail, or any of the two circles at the center. The edge is now colored red showing that it is selected. To delete the selected edge, click the Delete key.

4.10 Moving an edge

To change the edge's from-vertex to another vertex, click and drag the circle on the edge that is closest to the from-vertex. Drag the edge to the new from-vertex and release the mouse-button. Changing the to-vertex is analogues to changing the from-vertex.

4.11 Naming an edge

Each edge has a label. To assign or change the labe-text, click the label with the mouse. A blinking caret will display, allowing text to be entered with the keyboard. Once edited, press the Enter key to save the entered text. Alternatively, click the mouse outside the label to save the text. To cancel the editing, press the Escape button to restore the original label value. When the caret shows, Ctrl-C will copy the text onto the clipboard. Ctrl-C will replace the label-text with that on the clipboard.

4.12 Navigating across edges

When the from-vertex and the to-vertex of an edge is placed far away from each other in such a way that both do not fit within the window, then it can be difficult to follow the edge from one vertex to the other vertex. In such a case, click the arrow-head or arrow-tail to move the window over to the other vertex.

5 The OutOfBrain Virtual Machine (OOBVM)

The OutOfBrain Virtual Machine (OOBVM) is a background process which interprets and executes the graph. There are a number of graph-structures that are recognized by the OOBVM. These structures form the "grammar" or the "instruction-set" of OOBVM. Programming in OutOfBrain is therefore completely graph-oriented. The graph structures which are part of the instructions are collectively defined as the "instructions graph". The graph structures which are not part of the instructions are collectively defined as the "application graph". The instructions graph and the application graph both reside in the same graph. Edges which connect between instruction and application are called "interface edges".
Most label-texts used in the instruction set are enclosed by "<" and ">" characters (e.g. "<process>"). These label-texts are recognised by the OOBVM, so these are to be treated as reserved words. The programmer should therefore avoid using label-texts in the application graph that have either the "<" or the ">" characters. This chapter will describe the basic grammar of OutOfBrain.

5.1 <group>

The group structure is frequently used in OutOfBrain. It is essentially an isolation of a sub-graph. The group structure is mostly used as part of an instruction. A group is represented as a vertex labeled "<group>", having from-edges labeled "<group>" pointing to a number of vertices. These vertices are "members" of the group. See an example of a group structure in figure 5.1.

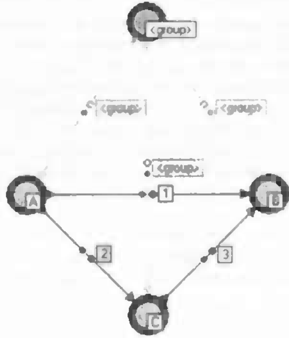


Figure 5.1 *The <group> structure*

All edges that exist between any of the group vertices are also considered members of the group. It is important to note that the members of any group are part of OutOfBrain's "instruction graph". Group structures are used a lot in OutOfBrain, and they can be cumbersome to construct manually. The OOBD supports a keyboard shortcut for constructing a group structure. Just select all vertices which are to be members of the group and press Ctrl-G.

5.2 <process>

Any vertex labeled "<process>" (unless it is a member of a <group>) represents a running process. Many processes can be added to an OutOfBrain application. The main loop running in OOBVM continuously searches for vertices which are labeled "<process>". For each <process>, the OOBVM determines whether it has a leaving-edge labeled "<run>". If such a connection exists, the corresponding to-vertex is "executed" as an instruction. After execution executed, the result of the instruction performed determines where the <run> connection should be moved to. All instructions have a result. For example, if the result of the instruction is "<true>", then the <run> connection is moved to the next instruction pointed at by "<true>". If the instruction is not recognised by the OOBVM, then the (default) result is "<next>". This is shown as an example in figure 5.2. If there is no edge found that corresponds to the result of te instruction, then the OOBVM determines whether there is an edge labeled "<next>". If this is the case, then this edge is followed instead.

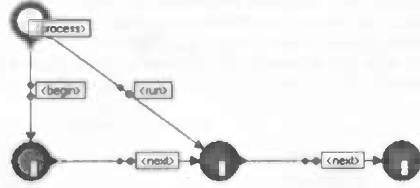


Figure 5.2 *The <process> structure executing an instruction*

If a <process> vertex does not have an edge labeled "<run>", OOBVM determines whether there is an edge labeled "<begin>". If such an edge exists, an edge labeled "<run>" is created with the same from-vertex and to-vertex as the "<begin>" edge. OOBVM then processes the "<run>" edge as previously described.
If the <process> does not have an edge labeled either "<begin>" or "<run>", then the process is "terminated" or "idle".
If there is no from-edge that matches the result of an instruction, then the <run> edge is moved back to the <begin>'s to-vertex. If there is no <begin> edge, then the <run> edge is destroyed and the process goes idle.
A process with a from-edge labeled "<begin>" is therefore a continuous running loop.

5.3 <branch>

The <branch> structure is similar to a "program block" in 'normal' programming. It has a leaving-edge labeled "<begin>" pointing to the <branch>'s first instruction to be executed. Initially an edge labeled "<run>" is created with the same from-vertex and to-vertex as the "<begin>" edge. The instruction pointed to by the <run> edge is executed, and the <run> edge is moved to the next instruction accordingly. The <branch> therefore acts as a <process>, except that it only runs a "single loop" when activated by a <process>. During execution of the <branch> structure, the <process> keeps its <run> edge pointing to the <branch> vertex. Once the <branch>'s <run> edge loops back to its <begin> instruction, then the <process> advances its own <run> edge to the next instruction pointed at by the edge labeled "<next>". The result of a <branch> is therefore always "<next>".

5.4 <execute>

An <execute> instruction is a reference to another instruction. The actual instruction to be executed is the to-vertex of the <execute>'s from-edge labeled "<run>". The <execute> instruction inherits the result of the actual instruction being executed.

5.5 <debug_pause> and <debug_step>

Any vertex labeled "<debug_pause>" (unless it is part of a <group>) will cause the OOBVM to pause. All <process> structures will halt execution until there is no vertex labeled "<debug_pause>". A vertex labeled "<debug_step>" has the same effect, except that the programmer can use the F8 key in OOBD to "step" the processes. Upon each press of the F8 key, all processes execute just one instruction (or branched instruction).

5.6 <implication>

The <implication> is the most-used instruction in OutOfBrain. It is comparable to a single-pushout "rewrite rule" in graph rewriting. It has a "left side" called "premise" and a "right side" called "conclusion". When an <implication> is executed, the OOBVM searches the application graph for patterns that match the premise. The pattern that is found is called the "redex". The labels of the edges and vertices of the redex must match the labels of the edges and vertices of the premise group. Labels of premise vertices and premise edges that are empty are treated as "wild-cards". When a redex is found, the redex is manipulated to complete the pattern represented in the conclusion. The <implication> can be thought of as a "search and replace" function. On another level of abstraction it can be thought of as an instance of causality.
The <implication>'s left and right side are represented as <group> structures. The <implication> has a from-edge labeled "<premise>", and a from-edge labeled "<conclusion>", respectively. Each edge points to a <group> structure.
The vertices that are part of the premise group are called "premise vertices", and vertices that are part of the conclusion group are called "conclusion vertices". Premise vertices may have from-edges labeled "<transition>" that connect to conclusion vertices. These edges indicate how the premise vertices map to the conclusion vertices.
When an <implication> instruction is executed, the OOBVM searches the application graph for sub-graphs that match the premise graph, with the additional condition that the redex must not resemble the conclusion. In other words, "find an instance of the premise which is not yet like the conclusion". If such a redex is found, then the implication is said to be in

the "before-state". Else, if a redex was found that matched the conclusion but not the premise, then the implication is in the "after-state". If a redex was found that matched both the premise and the conclusion, then the implication is in the "both-state". If a redex could neither be found for the premise or the conclusion, then the implication is in the "neither-state". The before situation has therefore the highest priority in OOBVM's search algorithm. If multiple redexes are found, then one is chosen at 'random'.
If the implication is in the before-state, then the OOBVM manipulates the redex so that it matches the conclusion. This phaze is called the "imply-state". These manipulations may include creating vertices, labeling vertices, destroying vertices, creating edges, moving edges, labeling edges and destroying edges. After applying these changes, the implication goes into the "implied-state".

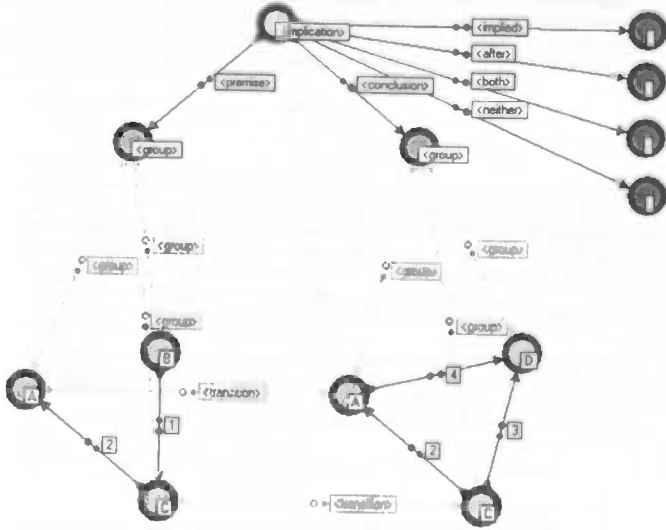


Figure 5.3 *The <implication> structure*

The result of the <implication> is "<implied>" for the implied-state, "<after>" for the after-state, "<both>" for the both-state, and "<neither>" for the neither-state. The result tells the <process> which path to follow to determine the next instruction to execute. Optionally, an edge labeled "<result>" can be connected from the <implication> to

another vertex. The result of the implication will be stored in the vertex's label after execution. This is useful during debugging.
In the example shown in figure 5.3, the implication in a before-state would alter the found redex by deleting the vertex labeled "B", delete the edge labeled "1", add a vertex labeled "D", add an edge labeled "4" between vertices "A' and "D", and add an edge labeled "3" between vertices "C" and "D". Important to note is that the labels of the vertices and edges in the redex must match those in the premise part. If a label in the premise part is en empty string, then this is treated as a 'wild-card'. In such a case, the corresponding redex vertex or edge may have any label.
Implication structures are used a lot in OutOfBrain, and they can be cumbersome to construct manually. The OOBD supports a keyboard shortcut for constructing an implication structure. Just construct a premise situation, select all the vertices and press Ctrl-I. And finally, alter the <conclusion> structure to achieve the desired rewrite rule.

5.7 <condition>

The <condition> instruction is very similar to the <implication> instruction.
Except for the instruction's label, the remaining structure is identical to that of
an <implication> structure. However, when a <condition> instruction is executed, it does not apply the rewrite rule when it results in the before-state.
Instead, it remains in the before-state and returns "<before>" as a result. The
<condition> instruction is therefore purely conditional.

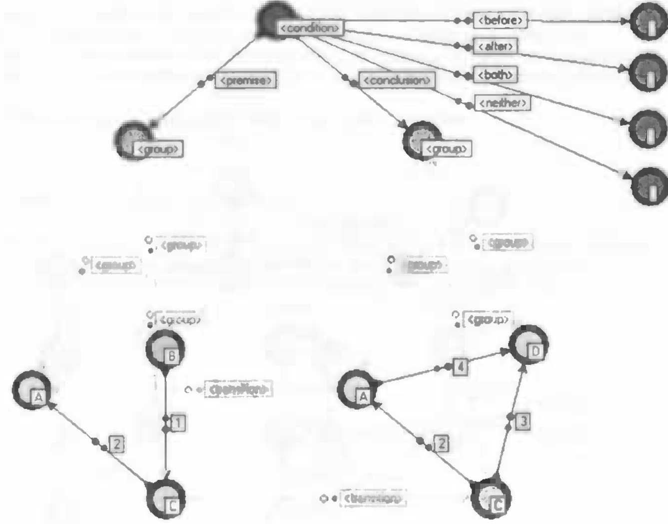


Figure 5.4 *The <condition> structure*

The <conclusion> group of a <condition> structure is optional. If the conclusion group is ommited, then the OOBVM simply determines whether a redex can be found that matches the premise group. If so, then the result is "<true>". Else the result is "<false>". A <condition> structure that has both a <premise> and a <conclusion> group is called a "double condition". A <condition> structure that has just a <premise> group is called a "single condition".

5.8 <wait>

The <wait> instruction is very similar to the <condition> instruction.
Except for the instruction's label, the remaining structure is identical to that of
a <condition> structure. However, when a <wait> instruction is executed, the <run> edge is not moved on to the next instruction until the condition returns either <after> or <both>. The result of the <wait> instruction is then <next>.

5.9 <candidate> edges

When the OOBVM searches the application graph for patterns that match the
premise part of either an <implication> or a <conclusion>, it does so in the
entire application graph and may find multiple redexes. In some cases we want
to implicitly tell the OOBVM that a particular premise vertex corresponds with a
particular application graph vertex. To do so, connect an edge from the premise vertex to the application graph vertex and label it "<candidate>". This
forces the OOBVM to use this "candidate vertex" in the redex, if such a redex exists. See figure 5.5 for an example.

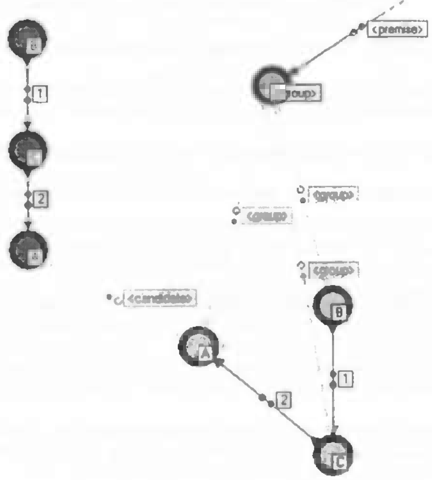


Figure 5.5 *The <candidate> edge*

Multiple <candidate> edges can be added. If a single premise vertex has multiple candidates, then the OOBVM searches for a redex where at least one of these candidates produce a match.

5.10 <reference> and <match> edges

When the OOBVM finds a redex for either an <implication> instruction or a <condition> instruction, then we might want to reference (parts of) this redex in a later instruction. This is especially the case where multiple redexes may be present in the application graph, and we want to focus on a particular redex for successive instructions . A <candidate> edge will not suffice here because we do not know beforehand which redex is selected. For such purposes the OOBVM supports the use of edges labeled "<reference>". The <reference> edge is very similar to the <candidate> edge, except that the <reference> edge points to a premise vertex of another instruction. Consider an instruction A that is followed by an instruction B. Further assume that there exists an edge labeled "<reference>" that points from a premise vertex PB in instruction B to a premise vertex PA in instruction A. When the OOBVM executes instruction A and finds a redex, an edge labeled "<match>" is created between vertex PB and its corresponding redex vertex RA. When instruction B is executed, then vertex RA functions as a candidate for finding PB.

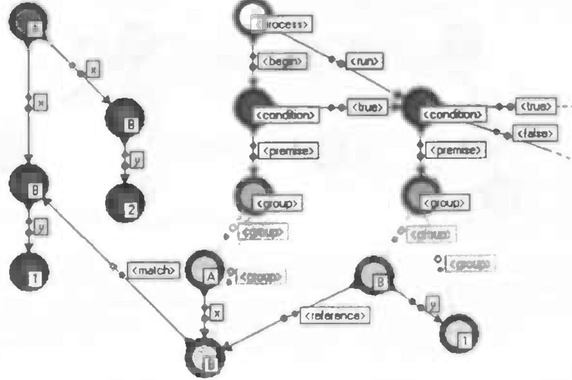


Figure 5.6 *The <reference> edge and <match> edge*

Figure 5.6 illustrates the use of the <reference> edge. The first instruction finds one of multiple possible redexes. The OOBVM detects the presence of a <reference> edge and creates a <match> edge to indicate the corresponding redex vertex. The second instruction then uses this redex vertex as the candidate.

5.11 Conditional operators

The OOBVM supports additional conditional operators that can be attached to the premise groups of instructions. These operators are used by the OOBVM when when it searches for a redex. A redex must meet the conditions specified by these operators. There are five different conditional operators, shown in table 5.1.

conditional operator	description
=	equals
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

Table 5.1 *The conditional operators*

To illustrate the use of a conditional operator, consider an instruction A with a premise group P where vertex PA is one of the premise vertices of P. An edge labeled "=" is connected from vertex PA to another vertex V that is not a member of P. Vertex PA has an empty label and V is labeled "5". The edge labeled "=" is a conditional operator. This conditional operator tells OOBVM that the redex vertex of PA must be labeled "5". For an edge to be interpreted as a conditional operator, the following conditions must be met:

1. The edge must be labeled either "=", ">", "<", ">=" or "<=".
2. The edge must be directed from a premise vertex that has an empty label.
3. The edge must be directed to a vertex outside the premise group.

The conditional operator points to a vertex whose label represents the conditional value. The label is interpreted either as a number, as a string or as empty. If the label is interpretable as a number, then the operator is applied as a numeric condition. If the label is not a number and is not empty, then the label is interpreted as a string and the operator is applied as a string condition. For numeric conditions the operators are self-explanatory. The compare operation for string conditions is based on the 8-bit ordinal value of each character.
If the label is empty then the conditional value is a calculated value. Calculated values are determined by the value operators. There are five different value operators, shown in table 5.2

value operator	description
=	is equal to
+	add
-	subtract
*	multiply
/	divide

Table 5.2 *The value operators*

A value operator is an edge that is labeled with one of the value operators listed in table 5.2. The edge can either point to a vertex within the premise group, or to another vertex outside the premise group.

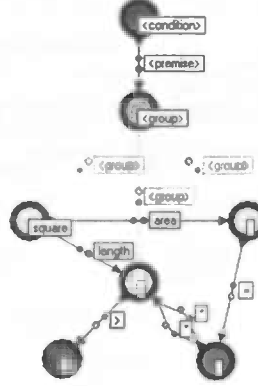


Figure 5.7 *Conditional operators and value operators.*

Figure 5.7 illustrates the use of conditional operators and value operators. The edge labeled ">" tells the OOBVM that the label of the redex vertex must numerically be interpretable as greater than zero. The edge labeled "=" indicates that "the area must equal (length * length)". Multiple value operators can be used to determine a conditional value. The following formula is applied when multiple value operators are used:

conditional value = (((sum("+") - (sum("-")) * (product("*"))) / product("/")

If any of the operator-types are not present, then they are eliminated from this expression. In general, value operators are only used for numeric calculations, and not applied as string operators.

5.12 Assignment operators

Asignment operators are very similar to conditional operators. Conditional operators are attached to premise vertices, but assignment operators are attached to conclusion vertices. The assigment operators are only used in conjunction with <implication> instructions. The assignment values are used to assign values to the labels of redex vertices when an <implication> instruction reaches the imply-state. There are five different assignment operators, shown in table 5.3.

assigment operator	description
=	set to
>	set greater than
<	set less than
>=	set greater than or equal to
<=	set less than or equal to

Table 5.3 *The assigment operators.*

To illustrate the use of an assignment operator, consider an <implication> instruction A with a conclusion group C where vertex CA is one of the conclusion vertices of C. An edge labeled "=" is connected from vertex CA to another vertex V that is not a member of C. Vertex CA has an empty label and V is labeled "5". The edge labeled "=" is an assignment operator. This assignment operator instructs OOBVM to set the label of the redex vertex that corresponds to CA to "5" during the imply-state of A. For an edge to be interpreted as an assignment operator, the following conditions must be met:

1. The edge must be labeled either "=", ">", "<", ">=" or "<=".
2. The edge must be directed from a conclusion vertex that has an empty label.
3. The edge must be directed to a vertex outside the conlcusion group.

The assignment operator points to a vertex whose label represents the assignment value. The label is interpreted either as a number, as a string or as empty. If the label is interpretable as a number, then the operator is applied as a numeric assignment. If the label is not a number and is not empty, then the label is interpreted as a string and the operator is applied as a string assignment. For numeric assignments, the assignment value is

set in such a way that the assignment operator would pass as a conditional operator. For example, when a redex vertex is labeled "4", but by means of assignment operators is required to be '>= 5', then the label of the redex is set to 5. But if in this case the redex were labeled "6", then the label would remain "6". For string assignments only the "=" operator is supported.
If the label is empty then the assignment value is a calculated value. These calculated values are determined in the same way as the calculated values are calculated for conditional operators, which is explained in the previous chapter.

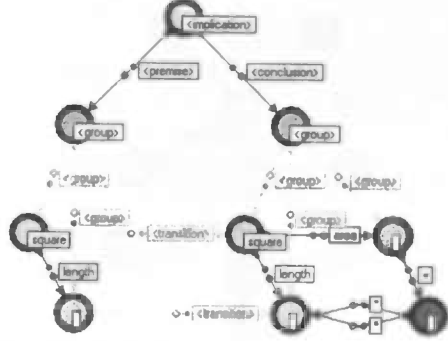


Figure 5.8 *The assignment operator.*

Figure 5.8 shows an example of an assignment operator. The assignment operator is the edge labeled "=". When the implication instruction reaches the imply-state, a new vertex is created and a new edge is created. The newly created vertex is labeled with the assignment value. The assignment value in this case is the "square of the length".

6 OutOfBrain Agents

OutOfBrain has built-in functionality for simulating autonomous agents and multi-agent systems. The type of agents that are modelled in OutOfBrain may not comply with the standard types of agents, and do not fit many of the very many different definitions of an agent. Instead, the agents in OutOfBrain are "graph-rewriting agents". These agents are run by the "Agent Manager" (AM) which is embedded within the OOBVM.

6.1 The Agent Manager

The Agent Manager (AM) is run within the OOBVM. The AM is activated when a <process> runs a vertex that is labeled "<agent>". The <agent> is parsed and executed by the AM. Each agent has a set of connected <implication> and <condition> structures which are loaded into the AM. These implications and conditions are called "branch-implications" and "evaluator-conditions", respectively.
The main component of the AM is the "Virtual Context Generator" (VCG). The VCG basically builds many copies of the entire graph and performs operations on each. Each of these graphs is called a "virtual context", which one can interpret as a "possible world". When an agent is run, the AM determines all the possible results of executing each branch-implication. There may be many different outcomes because every implication may find more than one redex. Each result is stored as a virtual context. This process is repeated for each virtual context recursively, resulting in a tree of virtual contexts constructed in breadth-first fashion. This tree is called the "Virtual Context Tree" (VCT). For every virtual context, the set of evaluator-conditions are evaluated. The results of these conditions tell the AM how "desirable" the virtual context, how "possible" the virtual context is, and whether the virtual context is permitted. Some branches may terminate, either because no branch-implications can be applied, or because a branch-implication may result in a virtual-context which is not permitted. When all virtual contexts are determined, or after a set time period, the AM determines the most desirable virtual context and records the sequence of branch-implications that were necessary to reach that virtual context. This sequence is called the "inference", and forms the "plan" of the <agent>. The inference is then stored as a sequence of instructions that are connected to <agent> vertex. The <process> structure can then execute this inference with the intention to transform the graph to match the desirable virtual context.

An <agent> is connected by edges to several <group> structures. The labels of these edges tell the agent what types of <group> structures they represent.

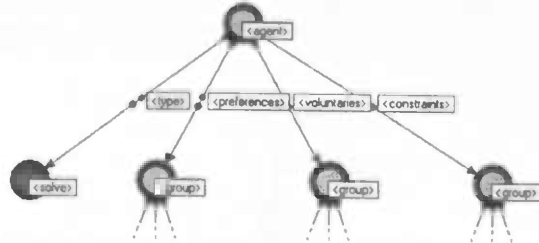


Figure 6.1 *The <agent> structure.*

The meaning of these different types of groups is explained in further detail in the following chapters.

6.2 <voluntaries>

The <voluntaries>-group contains a set of <implication> structures. The AM uses these to generate new virtual-contexts in the VCT, and are classified as branch-implications. A voluntary implication is executed in a virtual context, and the resulting graph is stored in a new virtual-context. If the implication cannot be applied, then the voluntary implication is ignored. The voluntary implications represent the actions the agent may or may not do in a given situation.

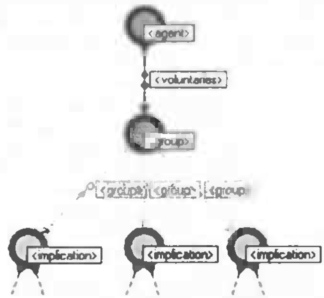


Figure 6.2 *An <agent> structure showing <voluntaries>.*

6.3 <involuntaries>

The <involuntaries>-group contains a set of <implication> structures. The AM uses these to generate new virtual-contexts, and are classified as branch-implications. An involuntary implication is executed in a virtual context, and the resulting graph is stored in a new virtual-context. If the implication cannot be applied, then the involuntary implication is ignored. The involuntary implications represent the agent's reflex actions that must be performed in a given situation.

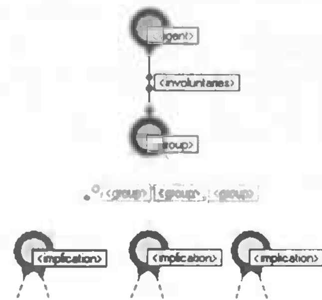


Figure 6.3 *An <agent> structure showing <involuntaries>.*

6.4 <constraints>

The <constraints>-group contains a set of <condition> structures. The AM uses these to determine whether a virtual context is permitted, and are classified as evaluation-conditions. The <condition> may either be a single condition or a double condition. If the constraint condition results in either <true> or <before>, then the virtual context is not permitted, and is removed from the VCT.

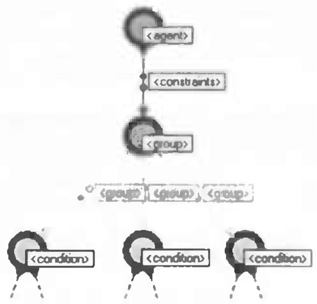


Figure 6.4 *An <agent> structure showing <constraints>.*

6.5 <consistencies>

The <consistencies>-group contains a set of <condition> structures. The AM uses these to determine whether a virtual context is consistent, and are therefore classified as evaluation-conditions. The <condition> may either be a single condition or a double condition. For a virtual context to be consistent, all consistencies conditions must either be <true> or <before>. If not, then the virtual context is removed from the VCT.

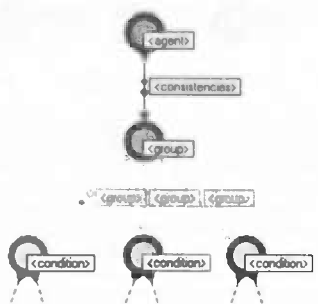


Figure 6.5 *An <agent> structure showing <consistencies>.*

6.6 <predictions>

The <predictions>-group contains a set of <implication> structures. The AM uses these to generate new virtual-contexts, and are classified as branch-implications. A prediction implication is executed in a virtual context, and the resulting graph is stored in a new virtual-context. If the implication cannot be applied, then the prediction implication is ignored. The prediction implications represent the agent's knowledge of how the "external world" will change in a given situation. Although prediction implications are branch-implications, they are converted to <wait> structures when used a part of an agent's inference.

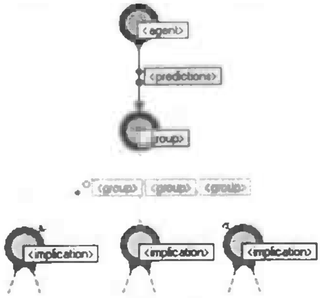


Figure 6.6 *An <agent> structure showing <predictions>.*

6.7 <preferences>

The <preferences>-group contains a set of <condition> structures. The AM uses these to determine how desirable a virtual context is, and are classified as evaluation-conditions. There are three different types of preference conditions, namely "<boolean>", "<count>" and "<obligation>". The type is stored in a vertex's label that is connected from the <condition> vertex with an edge labeled "<type>". When the <type> property is not present, then the type defaults to <boolean>. For boolean-preferences and obligation-preferences, the <condition> may either be a single condition or a double condition. For count-preferences, the <condition> must be a single condition.

The AM uses the <preferences> (together with <dislikes>) to rank the virtual contexts according to preference. For boolean-preferences, a virtual context has a high ranking if many conditions are either <true> or <after>. For count-preferences, a virtual context has a high ranking if many occurences of <true> situations (redexes) can be found. If obligation-preferences are present, then a virtual context is omitted from the ranking altogether if the condition result is either <before>, <both> or <false>. An agent's must achieve the obligations in any resulting inference.

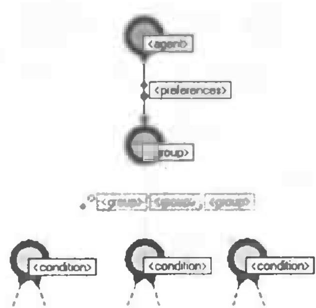


Figure 6.7 *An <agent> structure showing <preferences>.*

6.8 <dislikes>

The <dislikes>-group contains a set of <condition> structures. The AM uses these to determine how undesirable a virtual context is, and are classified as evaluation-conditions. The <dislikes> operate in the same way as <preferences>, except that <dislikes> make a virtual context have a low ranking. There are two different types of dislike conditions, namely "<boolean>" and "<count>". The type is stored in a vertex's label that is connected from the <condition> vertex with an edge labeled "<type>". When the <type> property is not present, then the type defaults to <boolean>. For boolean-dislikes, the <condition> may either be a single condition or a double condition. For count-dislikes, the <condition> must be a single condition.

The AM uses the <dislikes> (together with <preferences>) to rank the virtual contexts according to preference. For boolean-dislike, a virtual context has a low ranking if many conditions are either <true> or <before>. For count-preferences, a virtual context has a low ranking if many occurences of <true> situations (redexes) can be found.

6.9 The single-goal-agent

As an alternative to working with preferences and dislikes, a single goal is also supported. The single goal is represented as a <premise> and <conclusion> structure, that is connected directly to the <agent> vertex. The structure is therefore similar to a <condition> or <implication> structure, but is called the "goal-condition". When a single-goal-agent is run, it will first determine the value of the goal-condition. If the result is <before>, then the VCG is requested to build a VCT. The process will stop once an <after>-situation is found amongst one of the virtual contexts. The corresponding inference is then stored. The <before> situation is therefore the condition of the agent, and the <after> situation is the goal of the agent. The single-goal-agent can be used as a simple problem solver.

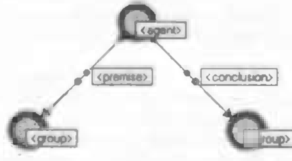


Figure 6.8 *The single-goal-agent.*

6.10 The agent types

There are two types of agents, namely "<active>", and "<solve>". The type is stored in a vertex's label that is connected from the <agent> vertex with an edge labeled "<type>". An active-agent will calculate and execute an inference. A solve-agent calculates an inference, but does not execute it. The result of an active-agent is "<achieved>" if the inference could be calculated and if the inference executed successfully. In all other cases the result is "<failed>". The result of a solve-agent is "<achieved>" if the inference could be calculated. In all other cases the result is "<failed>". A single-goal-agent has additional results "<both>" and "<neither>". These values are returned in the cases where the goal-condition results in "<both>" or "<neither>", respectively, because in these cases no attempt is made in calculating the inference. If the goal-condition is <after>, then no inference is calculated and the agent's result is simply <achieved>.

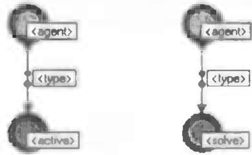


Figure 6.9 *An active-agent and a solve-agent.*

6.11 An example of an agent

To illustrate the use of an <agent>, an example is shown in the form of a simple problem solver. The puzzle is a well-known puzzle that is described as follows:

A farmer finds himself on one side (left side) of a river with a chicken, a fox, some grain and a boat. The farmer must take all these items to the other side (right side) of the river using the boat. The rules state, however, that the farmer may only take zero or one item in the boat. Furthermore, the fox and the chicken must not be left unattended, and neither should the chicken and the grain. What is the minimum order of actions that the farmer must perform to move himself and all the items to the right hand side?

We can quite easily solve this puzzle ourselves. There are two possible solutions:

Solution 1
1. Farmer takes chicken to right side.
2. Farmer travels to left side.
3. Farmer takes fox to right side.
4. Farmer takes chicken to left side.
5. Farmer takes grain to right side.
6. Farmer travels to left side.
7. Farmer takes chicken to right side.

Solution 2
1. Farmer takes chicken to right side.
2. Farmer travels to left side.
3. Farmer takes grain to right side.
4. Farmer takes chicken to left side.
5. Farmer takes fox to right side.
6. Farmer travels to left side.
7. Farmer takes chicken to right side.

To solve the problem in OutOfBrain, we start by making a model of the situation in the form of a graph. Here we describe the concepts of the puzzle that are relevant to the problem proposed. This will be our application graph (as opposed to the instruction graph).

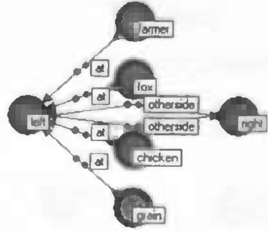


Figure 6.10 *The appligation graph.*

The application graph shows a simplified view of the first two sentences of the puzzle. Notice that the graph could have been named and structured differently, and still provide a good graph to work with. A "good graph" is loosely defined by two criteria:

1. "A graph that describes the essence of the static situation in its simplest form".
2. "A graph where relevant changes in the situation can be described in terms of simple graph operations".

A more formal definition of a "good graph" is difficult to determine because the two criteria show a trade-off relationship. Notice that we chose not to represent the "boat" in the graph, as it is not relevant to the essence of the puzzle. The boat is always with the farmer, so we do not need to represent the boat. Similarly, we do not have to represent the river. The puzzle is simply about a farmer, three distinct items and two distinct sides. It is easy to see why we need the "at" edges, because we need to be able to describe where the farmer and the items are. But one could argue that we could have left out the two "otherside" edges, because we can still describe the situation without them. The two "otherside" edges are there to meet the demands of the second criterium, for reasons that will become apparent later on. Constructing a good graph is usually an iterative process, tweaking the application graph during the design of the instruction graph. Making a good graph straight from the start requires practice with the type of graph rewriting used in OutOfBrain. The criteria for a good graph can be further extended with more requirements, such as "flexibility" and "re-usability", but these topics are beyond the scope of this text.

The next step is to create the instruction graph. The following paragraphs will describe each component of the instruction graph. Figure 6.11 shows a zoomed out view of the whole graph.

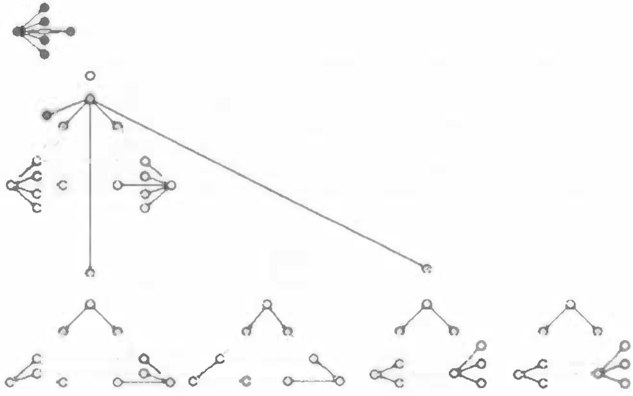


Figure 6.11 *Overview of the puzzle and the agent.*

In the zoomed out graph the application graph can be seen in the top left corner. The main <agent> structure is shown directly below the application graph, shown in figure 6.12. The agent is a solve-agent, as shown with the <type> edge. The agent is a single-goal-agent as it has <premise> and <conclusion> groups. The goal-condition shows that the premise matches the application graph, and therefore the goal-condition is in the <before> state. The agent would therefore attempt to calculate an inference if it were executed. Notice how the transition from the premise situation to the conclusion situation corresponds with what the puzzle asks us to solve.

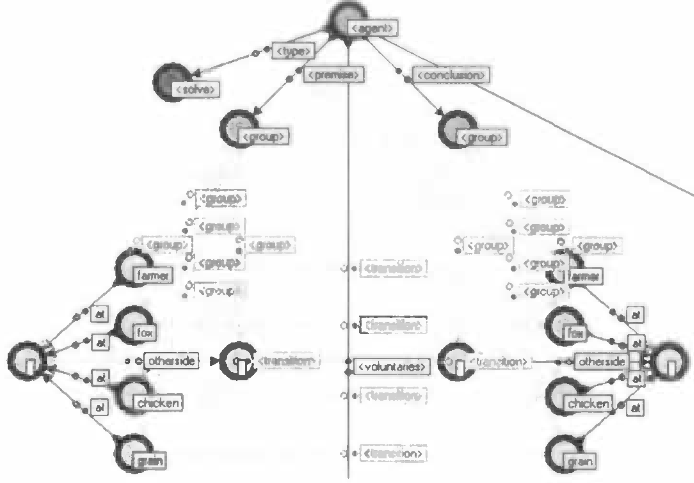


Figure 6.12 *The agent.*

The puzzle allows the farmer to go to the other side with just one item, or with no item at all. These two possible actions can be described as two voluntary implications. Figures 6.13 and 6.14 shows these voluntary actions respectively.

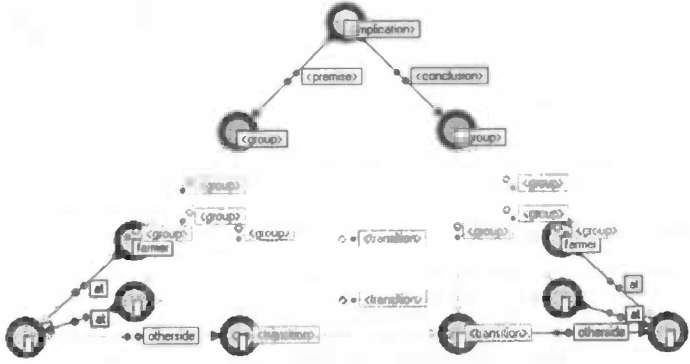


Figure 6.13 *Voluntary implication #1.*

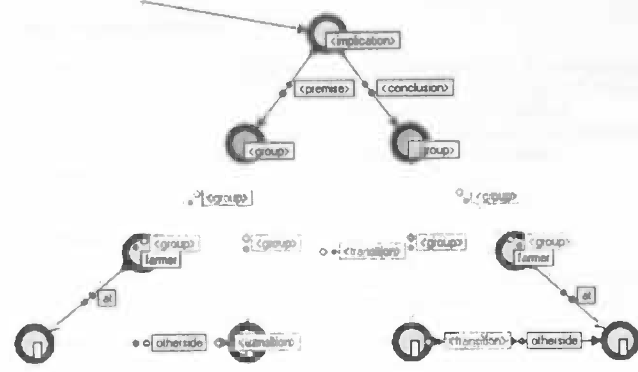


Figure 6.14 *Voluntary Implication #2.*

We have almost described the whole puzzle to the agent. The two remaining rules are two constraints, which state that the chicken and fox should not be left unattended, and neither should the chicken and the grain. We can describe these two constraints as constraint conditions. We need to describe these constraints as conditions in such a way that the <true> or <before> situation describes the constraint. Figures 6.15 and 6.16 shows how the before situation describes the absence of the farmer.

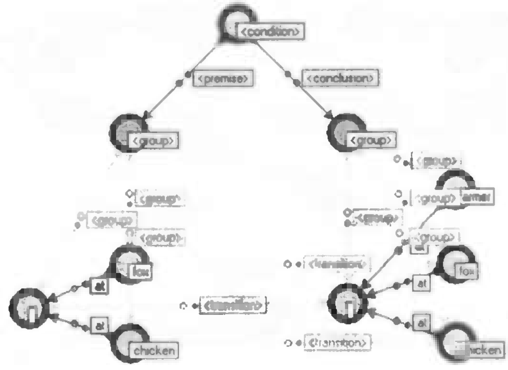


Figure 6.15 *Constraint condition #1.*

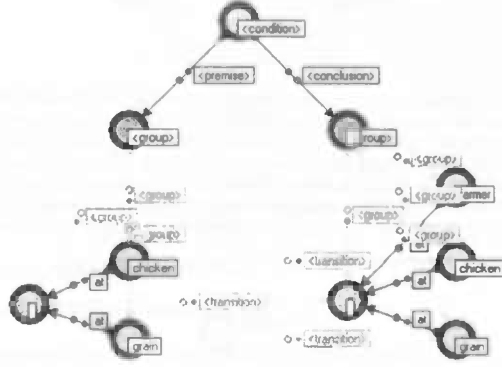


Figure 6.16 *Constraint condition #2.*

Representation of the entire puzzle is now complete. The agent has enough information to calculate an inference. If the agent is executed, then the agent will try many combinations of the voluntary implications to see which sequence will produce the desired result. When the inference has been found, the agent will print the sequence of instructions (implications) that were necessary to achieve the goal. And inference edge is connected from the <agent> vertex to the first instruction of the inference. As this is a solve-agent, the inference is not executed. The result is shown in figure 6.17. The instructions are shown from left to right, next to the <agent> vertex. Although not all visible, there are seven implications that describe the actions made by the farmer, which directly corresponds with one of the seven-step answers provided in solution 1 and solution 2. In fact, the agent is capable of finding both solutions, but just one of them chosen at random for the inference.

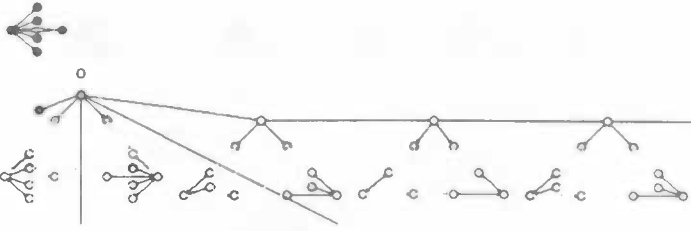


Figure 6.17 *Overview of the graph after execution of the agent.*

Figure 6.18 shows a closer view of the first instruction. It is visible that this resembles the first voluntary implication, except that the labels for "chicken", "left" and "right" are filled in. The agent does this ensure that the correct item is chosen.

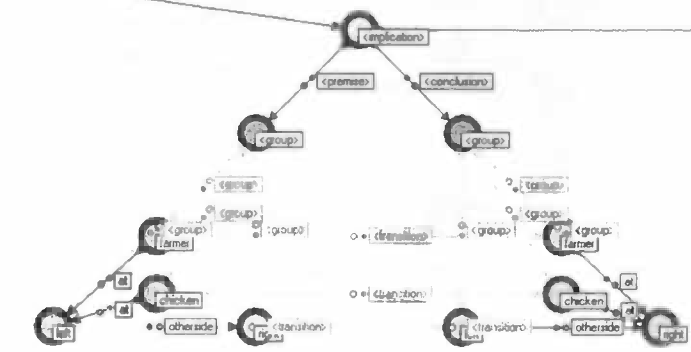


Figure 6.18 *The first instructions of the inference.*

If the agent were an active agent, then the inference would have been executed, moving the farmer with the items to the other side according to the instructions in the inference.

7 TCP communication

OutOfBrain supports structures for communicating via TCP. There are two basic structures for TCP, namely "<tcpserver>" and "<tcpclient>". They are not classified as "instructions" because they do not require a <process> to execute them. Instead, the TCP structures are run directly by the TCP manager. The following two chapters describe these instuctions.

7.1 <tcpserver>

The <tcpserver> structure is interpreted as a listening TCP server. Many <tcpserver> structures can be created, providing that each server listens to a different "port". A <tcpserver> structure is shown in figure 7.1. When the <status> property is set to "<enabled>" then a TCP server is created that listens to the port number that is set in the <port> property. If the creation of the server was not successful, then the <status> property is automatically set to "<disabled>". The server can be disabled by setting the <status> property to "<disabled>". The <status> property may be set by implications in the instruction graph.

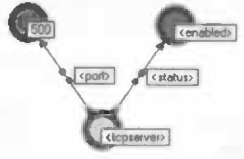


Figure 7.1 *The <tcpserver> structure.*

When a client connects to a <tcpserver>, a <client> structure is connected as shown in figure 7.2. The <address> property of the <client> shows the client's address. The <status> property of the <client> is set to "<connected>" while the connection persists. The <client> can be disconnected by setting the <status> property to "<disconnected>". When a <client> disconnects, the <client> structure is removed. Many clients may connect, and for each connection is <client> structure is created.

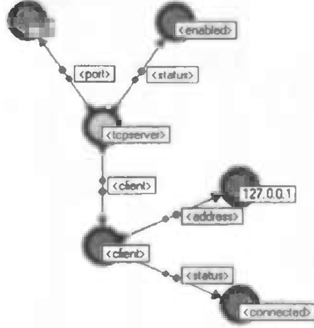


Figure 7.2 *A <tcpserver> structure with a connected client.*

7.2 <tcpclient>

The <tcpclient> structure is interpreted as a TCP client. Multiple <tcpclient> structures can be created. A <tcpserver> structure is shown in figure 7.2. When the <status> property is set to "<connect>" then a TCP client is created that connects to an address that is set in the <host> property, with the port number that is set in the <port> property. If the connection could be establisched successfully, the <status> property is set to "<connected>", else it is set to "<disconnected>". The client can be disconnected by setting the <status> property to "<disconnected>". The <status> property may be set by implications in the instruction graph.

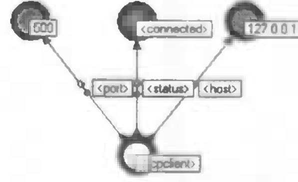


Figure 7.3 *The <tcpclient> structure.*

7.3 Sending and receiving data

The <tcpserver> and <tcpclient> both send and receive data in the same way. To send data from a <tcpclient> to its connected host, first create a string of vertices that are interconnected by edges labeled "<next>", directing from the start to the end of the string to be sent. In the labels of each vertex of this string, enter a piece of text. The last vertex in the string should be labeled "<newline>". Next, connect an edge from the <tcpclient> to the first vertex of the string and label it "<send>". At this moment, the TCP manager will detect the presence of the <send> edge, and will determine whether a string of veryices is connected that ends with a <newline> vertex. Then the string will be removed, and the textual version is sent to the host. The data that is actually sent is a string of texts, separated by space characters, and ending wit a #10 byte character. Similarly, if a <send> edge is connected from a <tcpserver>'s <client> vertex to such a string of vertices, then this text is sent to the corresponding client.

When a <tcpclient> receives data from its connected host, it collects all data until a #10 character is read. The data is interpreted as text, and is separated into smaller pieces of text where whitespace is used as a delimiter. These pieces of text are then stored into the labels of a string of vertices. An edge labeled "<received>" is then connected from the <tcpclient> vertex to the first vertex of the string. Similarly, a <client> of a <tcpserver> receives data in the same way.

The easiest way to illustrate this, is to create a <tcpserver> structure and a <tcpclient> structure in the same graph, and let the <tcpclient> connect to the <tcpserver>, as shown in figure 7.4. The figure shows that the client has received a string of data. The data can be sent back to the server by renaming the "<received>" edge to "<send>".

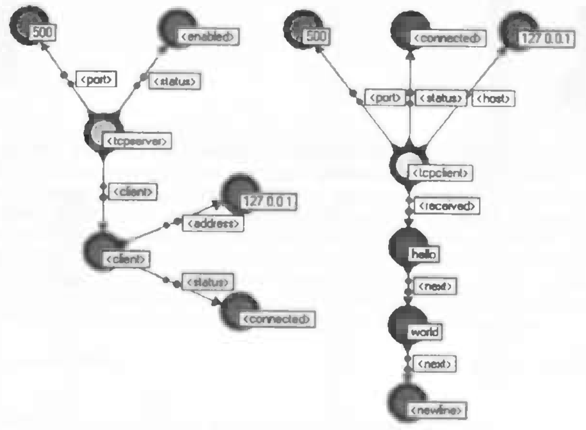


Figure 7.4 *A <tcpclient> connected via TCP to a <tcpserver>.*

Appendix B: Overview of todo-agent and user-agent

Below are the overviews of the todo-agent's and the user-agent's implementation. The diary-agent is provided in Section 5.2.

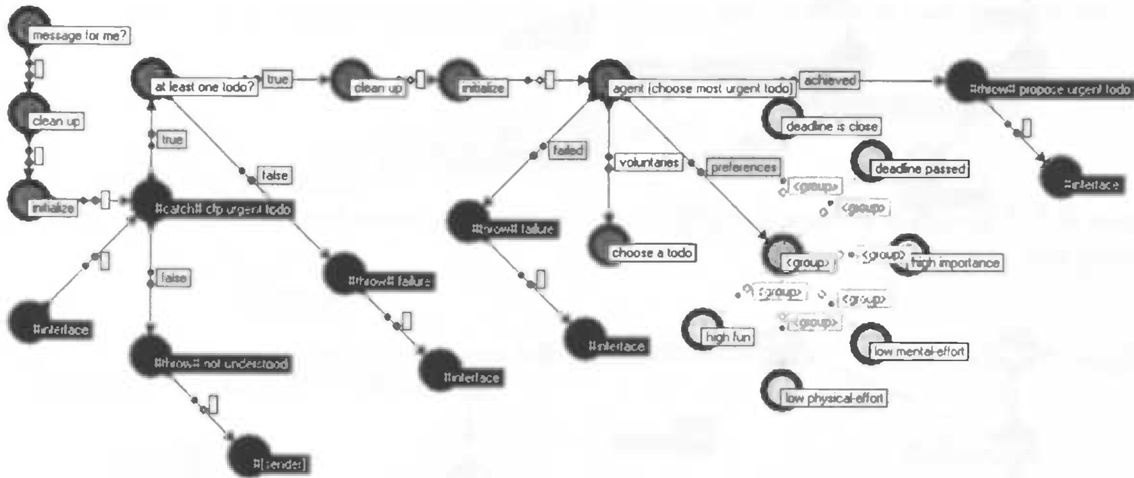


Figure B.1: The todo-agent, represented in OutOfBrain-pseudo-code.

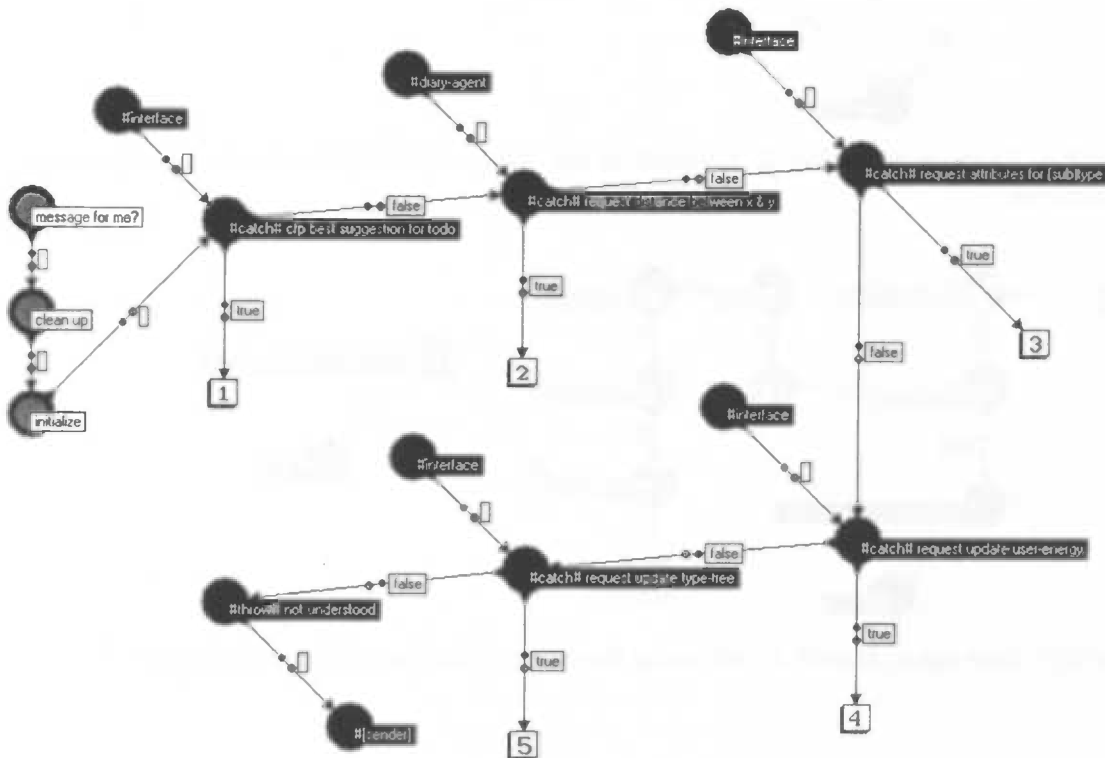


Figure B.2: The user-agent, represented in OutOfBrain-pseudo-code.

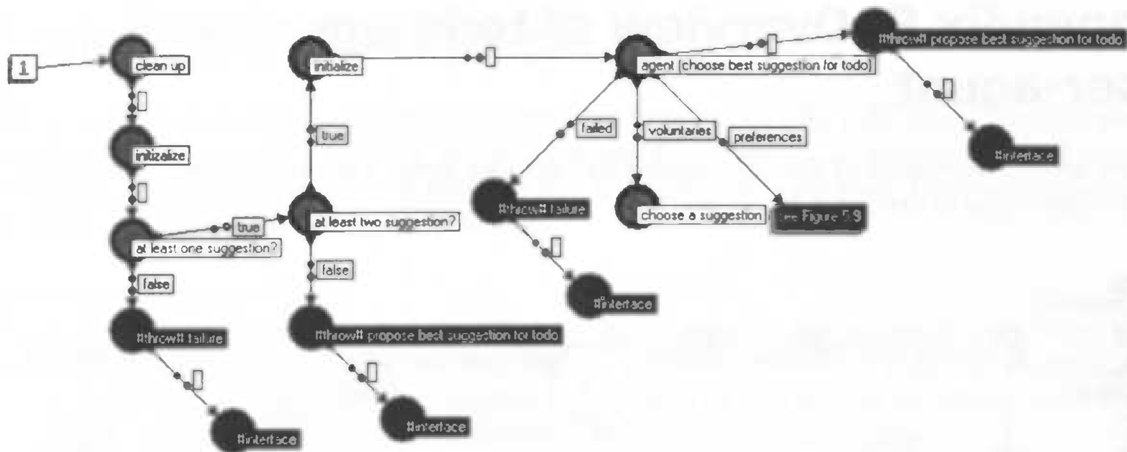


Figure B.3: User-agent branch 1; selection of the best suggestion from a list provided by the diary-agent.

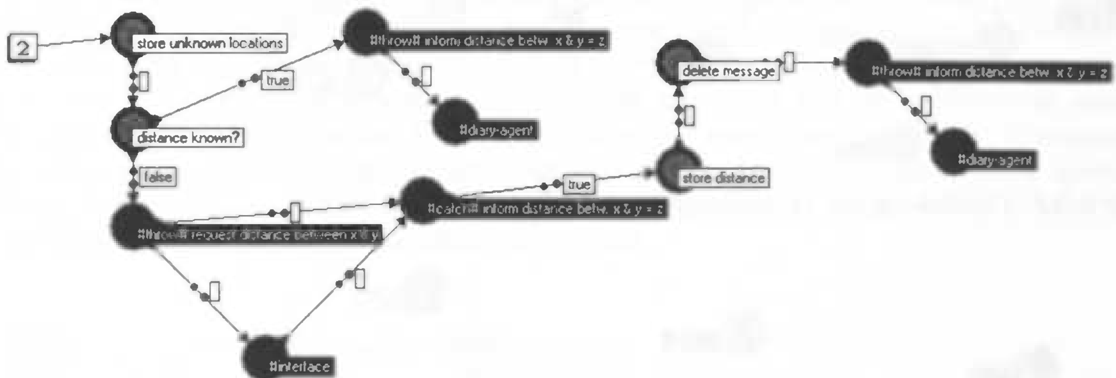


Figure B.4: User-agent branch 2; retrieval of the distance (travel time in minutes) between two locations.

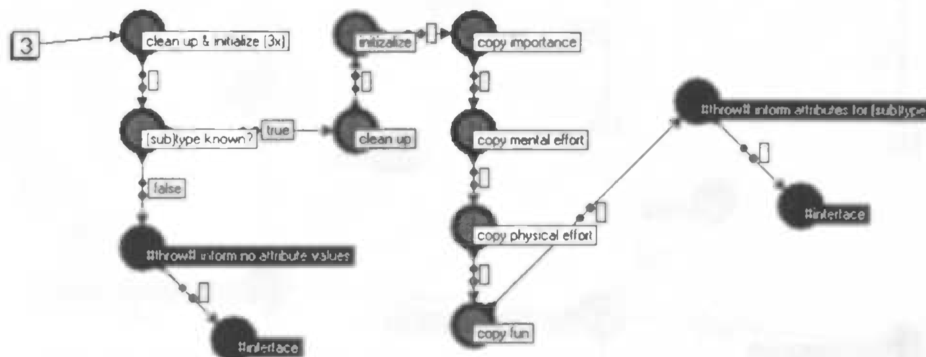


Figure B.5: User-agent branch 3; retrieving the average attribute values for a given subtype.

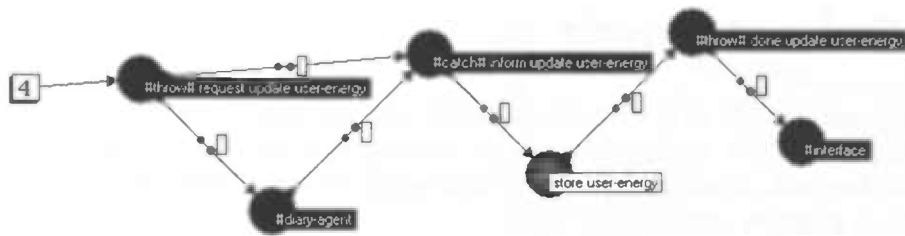


Figure B.6: User-agent branch 4; updating the user-energy (also known as average-energy).

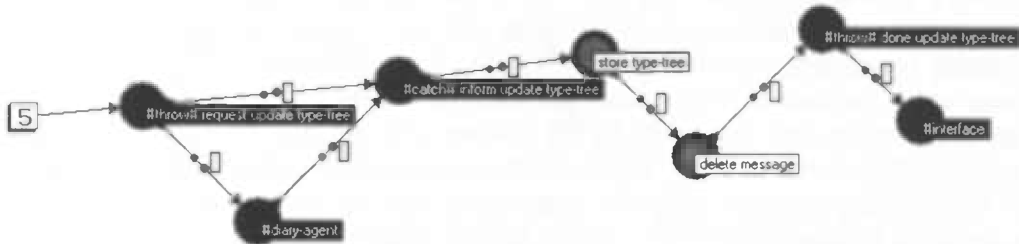


Figure B.7: User-agent branch 5; updating the type-tree.

Appendix C: host graph

The core of the host graph consists of a diary, the do-items and the todo-items. There is also an abundance of supporting data in the host graph: messages, a user model and some reference objects. We start with an example of a todo-item and a do-item. Then we provide an example message and an example user-model.

The representation of items

Figure C.1 shows a typical todo-item. The user has to provide this information through the interface, using buttons, checkboxes and slides[1]. The task is to write a project proposal, it takes approximately four hours, the deadline is the fourth of July and the location is the IDEA building in Groningen. The item is of type work and subtype writing.

The attributes on the left show that we are dealing with an important task that takes tremendous mental effort, but hardly any physical effort. Furthermore, this user clearly likes writing his or her project proposals. This item is probably high on the todo-agent's urgency list, since the deadline is close (given that today is July the second). Furthermore, the task is fun, important and not physically demanding. The reason that it is probably not the topmost todo-item on the urgency-list is that it takes four hours and a lot of mental effort to complete. If this item is chosen to be moved to the diary, the user-agent will try to satisfy as many preferences as possible (Section 5.4). Travel times between items are to be minimized. Diversity per day should be maximized, with respect to type, subtype, importance, mental effort, physical effort and fun. Furthermore, we want the item to be scheduled as soon as possible and finally, the deadline is taken into account.

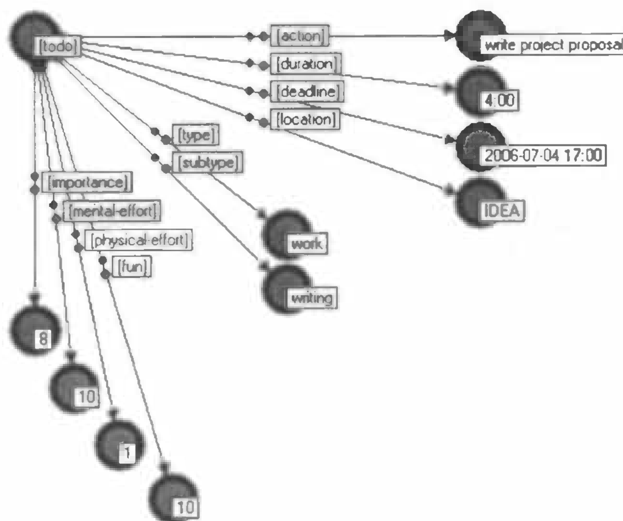


Figure C.1: OutOfBrain representation of a typical todo-item.

As can be seen from Figure C.2, a do-item is almost identical to a todo-item. The difference is that a do-item must have a start-time and an end-time[2] and that it is always connected to a day in the diary.

[1] Instead of providing the attributes on the left of Figure C.1 manually, they can be estimated automatically using statistical analysis of other items with the same type and subtype.

[2] Do-items can also have a deadline, although the one in Figure 5.3 does not.

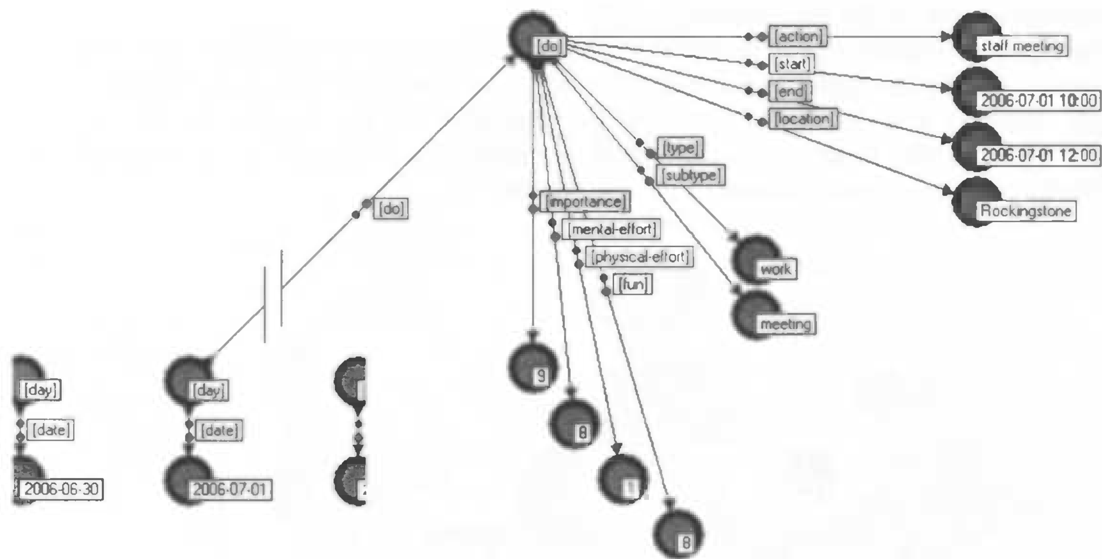


Figure C.2: OutOfBrain representation of a typical do-item, connected to a day in the diary.

The representation of messages

Below, in Figure C.3, is an example message. The todo-agent has selected the most urgent todo-item and he now proposes it to the interface. The todo-agent's choice is not evaluated, instead, the item is directly highlighted on the user-interface.

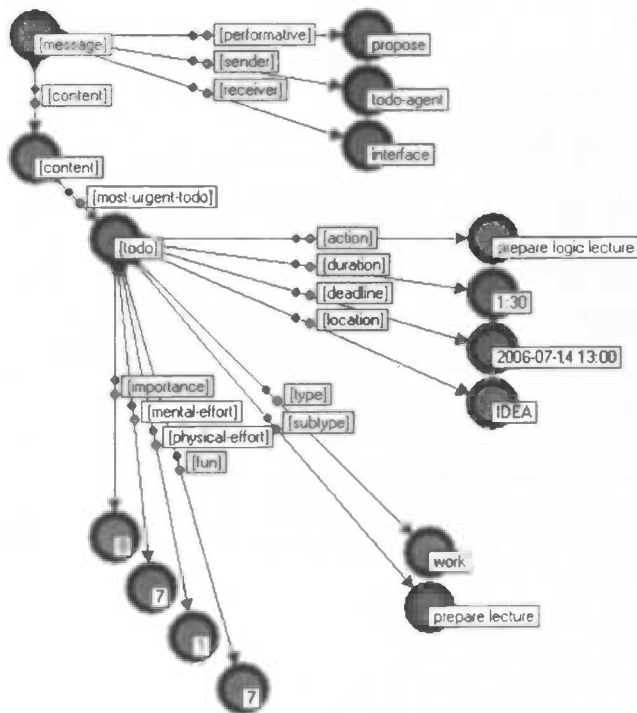


Figure C.3: A typical message. The todo-agent tells the interface which todo-item is the most urgent.

The representation of the user-model

The user-agent maintains a small model of the user and the world around him or her. The first component of this user-model, the distance-matrix, is depicted in Figure C.4. This particular example contains four locations. The user is prompted for the distance (travel-time in minutes) between two locations as soon as it is needed for planning. In this example, all possible distances are already known to the user-agent.

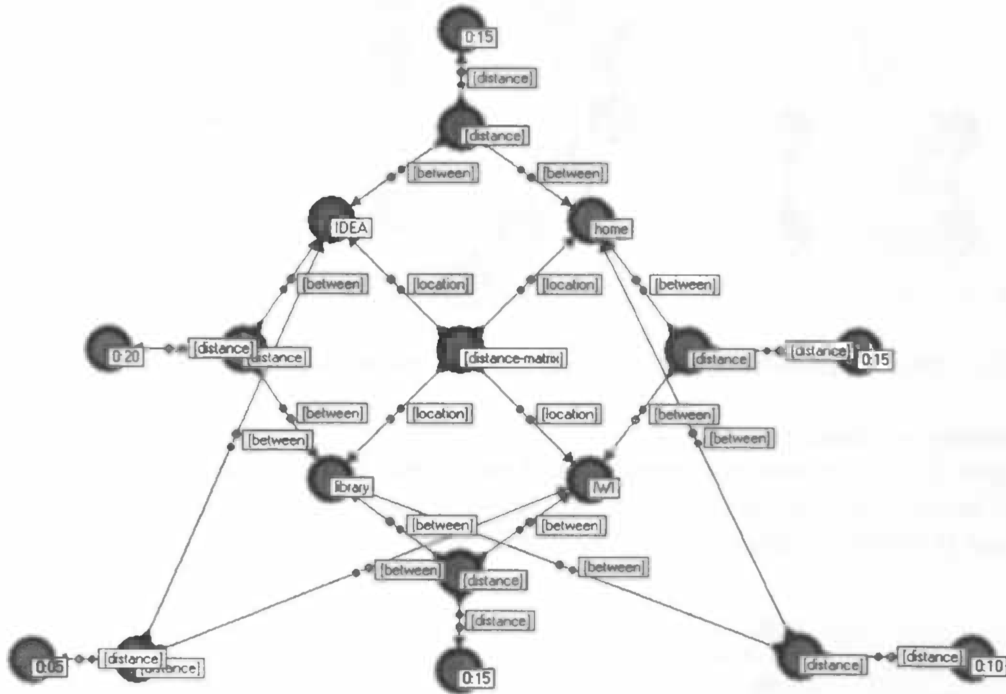


Figure C.4: An example distance-matrix with four locations in it.

The second part of the user-model is the type-tree (Figure C.5). It contains all type/subtype pairs occurring in the diary. The values of 'importance', 'mental-effort', 'physical-effort' and 'fun' are average values derived from all the do-items of the same type and subtype. The attributes of new items of a known type/subtype are set to the appropriate averages. The user can then augment them manually, but the averages should approximate the desired values, especially after long term use.

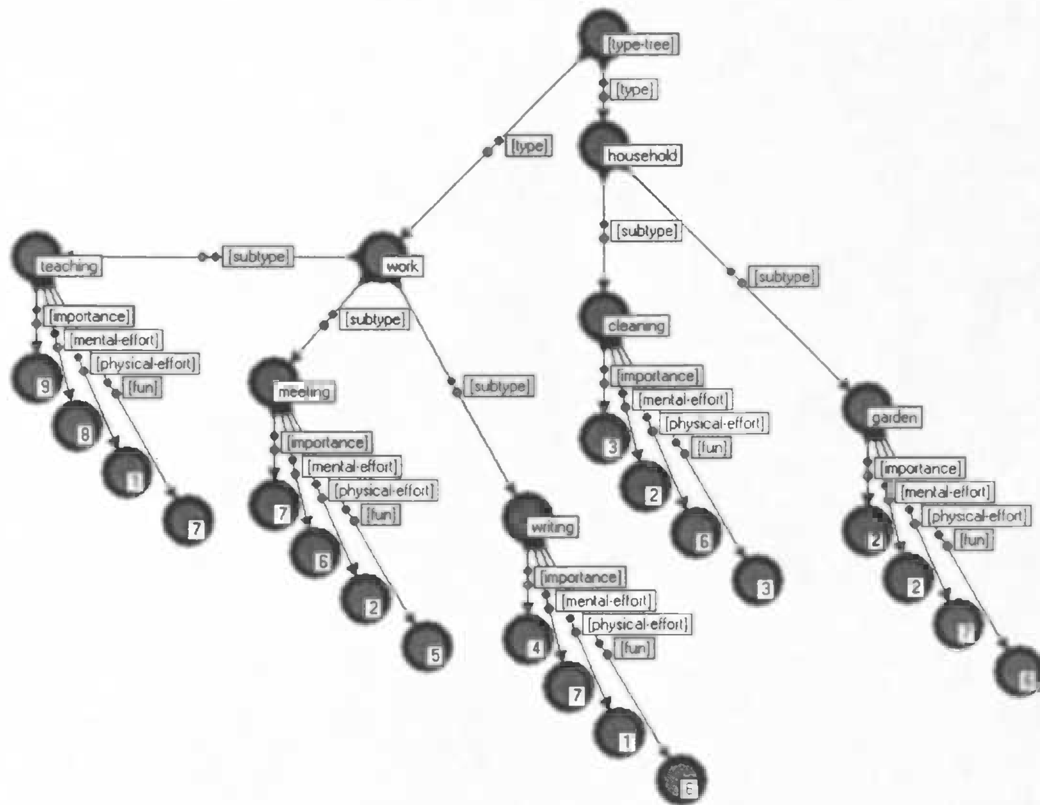


Figure C.5: An example type-tree containing the average attributes for two types and five subtypes.

The third and final part of the user-model is the average-energy (also known as user-energy). After using the system for twenty four days, the average-energy could look like the one in Figure C.6. It consists of two parts. First, the diary-agent checks whether this part of the user-model is already up-to-date, hence the attribute 'last-date'. If it is not, he proceeds with analysing the diary day by day, until 'last-date' is equal to yesterday. The values 'total-importance', 'total-mental-effort', 'total-physical-effort' and 'total-fun' are sums over all the items analysed so far.

These sums are divided by the number of days that were analysed so far (twenty four), resulting in the average amount of importance, effort and fun per day. These averages are then rounded to integer values. Unfortunately, this third part of the user-model was not used in any of the preferences of the system due to practical reasons. The user-agent was supposed to use it while choosing a suggestion from the list provided by the diary-agent. The user-agent could then try to obey the user's energy-threshold when planning todo-items. Another possibility is to ensure a minimum amount of fun per day (if this can be unified with all the other preferences).

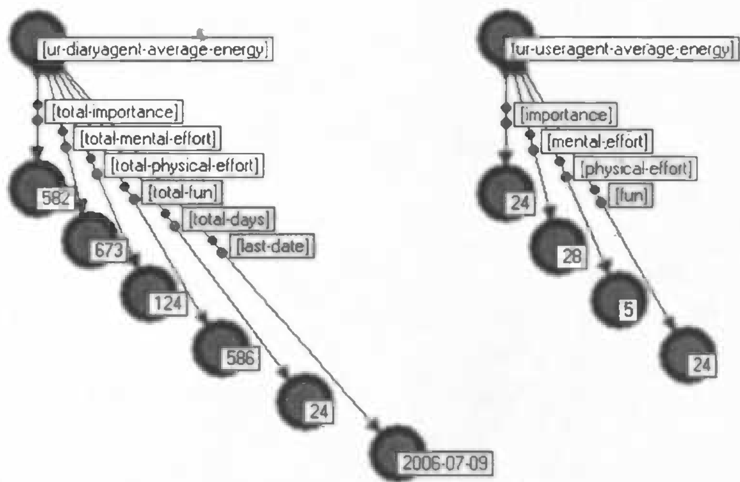


Figure C.6: An example of the user-energy part of the user-model. Twenty four days were analysed.