PROGRAMMING RATIONAL AGENTS

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy

by

Benjamin Hirsch

June 2005

Department of Computer Science
University of Liverpool
To my old and to my new family
Abstract

This thesis tackles the implementation, application, and formalisation of a logic-based programming language for rational agents. This is based on the previous, essentially theoretical, work on METATEM. Our work here comprises three major aspects. Firstly, we provide a practical implementation of the agent programming language METATEM, extended with abilities and beliefs. Second, we extend both the implementation and application to encompass simple, but powerful grouping structures. We show how a number of sophisticated applications can be naturally developed based on these rational agent groups. Finally, in addition to providing operational semantics for both single and multi-agent implementations, we show how the grouping structures correspond closely to complex process calculi for mobile agents.
Acknowledgements

While many people have been instrumental in the creation of this thesis, I wish to thank first and foremost my advisor Michael Fisher. Not only has he put his faith in me by allowing me to work with him for more than three years, but he also has had nothing but encouragement and patience for me during that time. Chiara Ghidini, my second advisor, has been an inspiration and support, during her time in Liverpool and far beyond that. This thesis is their work as much as it is mine (apart from any mistakes, which are solely mine).

The University of Liverpool, has provided for me for all this time, with material support, but even more so an environment of dedicated and brilliant minds that welcome every opportunity for discussion and advice. While fire regulations might require doors to be closed at all times, I always had the feeling that people kept them wide open.

There is not enough space here to acknowledge everyone that has been important to me and this work in this space. Andrea lit my life and made me the happiest PhD student alive by marrying me, and staying by me through all this time. My parents and siblings provided moral and practical support, and most importantly never doubted my decisions or abilities, which is more than I can say for myself.

Good friends kept me sane and ensured that life consisted of more than just work. You all know who you are. Finally, I would like to thank Gene for his (almost) endless supply of 42 minutes of fun.
# Contents

Abstract .......................... v  
Acknowledgements ................... vii  

1 Introduction ........................ 1  
1.1 Motivation .......................... 1  
1.2 Agents and Rational Agents ........... 2  

I Single Agents ...................... 11  

2 Background — Logic and Agents ........... 13  
2.1 Modal Logics .......................... 13  
2.2 BDI ................................... 15  
2.3 Default Logics ......................... 20  
2.4 Situation Calculus ...................... 23  
2.5 KARO ................................ 27  
2.6 Logic Programming ...................... 33  

3 Single METATEM Agents ................. 39  
3.1 Temporal Logic .......................... 39  
3.2 METATEM ............................... 45  
3.3 Implementing METATEM ................. 54  
3.4 Structural Semantics for METATEM ....... 64  

II Multiple Agents .................... 89  

4 Background — Multi Agent Systems ... 91  
4.1 Concurrency ........................... 91  
4.2 Communication .......................... 93  
4.3 Multi-Agent Formalisms ................. 95  
4.4 Multi-Agent Behaviours ................... 100  
4.5 Multi-Agent Frameworks ................. 102
5 Groups of \textsc{metatem} Agents

5.1 Structured Agent Space .......................... 108
5.2 Object Oriented Programming ...................... 112
5.3 Structure and Information ........................ 115
5.4 Manipulating the Agent Structure ................ 117
5.5 Group Policies .................................. 126
5.6 Implementing Groups of \textsc{metatem} Agents ........ 129
5.7 Structural Semantics for Multiple \textsc{metatem} Agents ...... 131

III Applications .................................. 143

6 Single Agent Examples ............................. 145

6.1 Backtracking .................................. 145
6.2 Bounded Belief ................................ 147
6.3 Deliberation .................................. 152

7 Multiple Agent Examples ............................ 157

7.1 Coffee Example ................................ 157
7.2 Museum Example ................................ 161
7.3 Travel Agent Example ............................. 171

8 Related Work .................................... 175

8.1 Holonic Agents ................................ 175
8.2 Group Based Approaches ........................... 176
8.3 Agent Frameworks ................................. 179
8.4 Milner's Bigraphs ................................ 180

9 Evaluation ........................................ 195

9.1 Conclusions ................................... 195
9.2 Future Work ................................... 197

Bibliography ........................................ 199

Appendices .......................................... 216

A Documentation .................................... 217

A.1 System Description ............................... 217
A.2 Predicates .................................... 218
A.3 Rules ......................................... 220
A.4 File Types .................................... 222
A.5 External Abilities ................................. 225
A.6 Internal Predicates ............................... 226
A.7 Default Constructs ............................... 228
## CONTENTS

B Code

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1 Default File</td>
<td>231</td>
</tr>
<tr>
<td>B.2 Coffee Example Code</td>
<td>235</td>
</tr>
<tr>
<td>B.3 Museum Example Code</td>
<td>239</td>
</tr>
</tbody>
</table>

List of Figures 244

List of Definitions 246

Index 248
Chapter 1

Introduction

1.1 Motivation

The aim of this work is twofold. On the one hand, we aimed at providing a full and practical implementation of MetateM [Fis94, Fis95] and extensions [FG99, FK99, FG02a], which has been suggested for years as being a language well suited to bridge the gap between logic-based specification of multi-agent systems and the actual implementations thereof. Secondly, we investigated the use of structured agent spaces in programming non-trivial multi-agent systems. Parts of this work have already been published [HFG03a, HFG03b, FHG04, FGH04, HFGB05], mostly concerning the structuring of multi-agent systems and potential applications of this approach.

The structure of this thesis is as follows. After introducing the problem area we will start by establishing our view of what constitutes an agent. We continue by surveying some of the more important logics and languages for (single) rational agents in Chapter 2. Theories of multiple agents are described in Chapter 4, while the main work of this thesis is divided between Chapter 3 (for single agents) and Chapter 5 (for the multi-agent case). Chapters 6 and 7 present some practical examples of MetateM programs, again divided between single and multiple
agent examples. Finally, we compare our work with some of the research that has been done already (Chapter 8), in particular providing a translation to, and from, mobile process calculi. We conclude with an evaluation in Chapter 9.

1.2 Agents and Rational Agents

In this section we discuss definitions of an agent and introduce the reader to the connection between agents and logic. This section follows closely the introduction of [Woo94].

1.2.1 What is an Agent?

The Merriam-Webster Dictionary [Web98] entry for Agent is:

<table>
<thead>
<tr>
<th>Function</th>
<th>noun</th>
</tr>
</thead>
<tbody>
<tr>
<td>Etymology</td>
<td>Middle English, from Medieval Latin agent-, agents, from Latin, present participle of agere to drive, lead, act, do; akin to Old Norse aka to travel in a vehicle, Greek αγόν to drive, lead</td>
</tr>
<tr>
<td>Date</td>
<td>15th century</td>
</tr>
<tr>
<td>1</td>
<td>one that acts or exerts power</td>
</tr>
<tr>
<td>2</td>
<td>a: something that produces or is capable of producing an effect: an active or efficient cause b: a chemically, physically, or biologically active principle</td>
</tr>
<tr>
<td>3</td>
<td>a means or instrument by which a guiding intelligence achieves a result</td>
</tr>
<tr>
<td>4</td>
<td>one who is authorised to act for or in the place of another: as a: a representative, emissary, or official of a government &lt;crown agent&gt; &lt;federal agent&gt; b: one engaged in undercover activities (as espionage): SPY &lt;secret agent&gt; c: a business representative (as of an athlete or entertainer) &lt;a theatrical agent&gt;</td>
</tr>
</tbody>
</table>
1.2 Agents and Rational Agents

Although this is not a definition coined for agents in Computer Science, we can see several important notions here: *acting* and *producing an effect*. Also, a representative has *autonomy*, can act on her own will. If we now translate the above into Computer Science speak, we arrive at

**Agent:** A (*temporally continuous*) computer system, situated in some environment, that is capable of *flexible*, (responsive, pro-active, social) *autonomous action* in order to meet its design objectives.\(^1\)

In this definition, the agent must be continuous — a one-time computation, however intricate, does not qualify as an agent. However, an agent does not need to run continuously; it can go to sleep, or even terminate itself, but the state it is in must then be recorded somehow. Furthermore, it must be able to take action on its own. A word-processor might act responsively by showing every typed letter on the screen, but that does not qualify as autonomous action. Of course an agent can exhibit some reactive behaviour, but not solely — it must be able to act on changes of the environment. Very often, communication (between agents, but also between humans and agents) is considered essential. Using its communicative abilities, an agent can, for example, “out-source” parts of its task. This entails that it must be able to “talk” to other agents, that is, try to call upon other agents to assist it in accomplishing its goal. Note that this goes further than merely exchanging information or running processes in some distributed manner, since agents do not need to comply with requests, or can engage in some form of negotiation. In fact, this is one of the features that distinguishes agents from objects (in the object-oriented-programming sense), that “objects do it for free, agents do it for money”[JSW98]. Having said all this, there still is no generally accepted definition of an agent, and some researchers would not accept one or more of the above propositions, or would want to add others — some, for example,

\(^1\)This definition is a combination of definitions found in [Sho90, JSW98, Woo94].
consider the ability to learn essential for an agent (see for example [GSH+97]).

1.2.2 A Bit of History

Research into agents originates in different fields, such as Artificial Intelligence, object-oriented programming, and human computer interaction. The first application that had some relation to agents as we understand them today was Newell and Simon’s planning system GPS, developed in 1961 [NS61]. They formulated the physical symbol system hypothesis. A physical symbol system is defined to be a set of physically realisable entities, or symbols, that can be combined to structures, and which is capable of running processes on those structures. The hypothesis says that such a system is capable of general intelligent action. In less technical terms, the general idea behind the symbolic approach is that the environment can be divided in discrete elements, and that it is possible to find rules that apply in this discrete model of the world. The approach has two major problems to solve [Woo94]:

- the transduction problem: that of translating the real world into an adequate, accurate symbolic representation of the world, and to do so in time for that description to be useful.

- the representation/reasoning problem: that of representing information symbolically, and getting agents to manipulate/reason with it, in time for the results to be useful.

After initial hype, it quickly became clear that modelling intelligent behaviour was certainly not as simple or as straightforward as many had hoped. Planning from first principles, as the technique is called, implies that systems have to devise a full plan from scratch to reach a given goal. Moreover, the planning algorithms are predicated on the assumption of calculative rationality, meaning
that they compute the optimal solution — if one exists. Whilst planning intelligent behaviour in an extremely simplified environment (like the blocks-world) already proved to be hard, coping with the real world was next to impossible — there are just too many parameters to take into account and because the machines were trying to find the best rational action, they basically did nothing but “think” about what to do — if they did not crash because of lack of memory. In short, Newell and Simon’s approach of first-principles planning was too slow, not scalable, and was later shown to be NP hard (for details we refer to [Cha87]), meaning that not only does the computation usually take a long time, but that there is no way to make an efficient algorithm.

Alternatives to the symbolic paradigm were proposed. Rodney Brooks, one of the more distinguished critics of symbolic AI, developed what he called behavioural AI [Bro86]. While the symbolic approach tried to build reasoning systems that basically computed the impact of each possible action on the environment by applying some form of search, Brooks’ approach was to use simple input-output rules or behaviours. As opposed to symbolic approaches, behavioural approaches do not have a (symbolic) representation of the outside world, and no search is performed. Instead, an agent consists of several behaviours that compete against each other. This is also called the subsumption architecture. The interested reader is referred to [Bro91], which gives an overview of the subsumption architecture and its underlying philosophy. The technique, while impressive, also has its drawbacks. It is very hard to predict the impact of new rules on the system — the interaction between the different layers becomes so complex that developing such a system becomes guesswork, and debugging such a system moves from difficult to impossible.

As so often, a mixed approach seems to work best. In this hybrid approach, typically three layers are used [JSW98]. At the lowest level there is a “reactive”
layer, which makes decisions based on raw sensor input. Often, this layer is implemented using Brooks’ subsumption architecture. The middle layer abstracts away from raw sensory input and deals with a knowledge level view of the agent’s environment [New82]. Typically, this layer makes use of symbolic representations. The uppermost layer tends to deal with social aspects of the environment — typically other agents in the environment are represented here and interaction between agents is controlled via this uppermost layer.

The last view of agents we want to present here, and which will be the backdrop to the remainder of this work, is that of practical reasoning agents. It is inspired by the theory of practical reasoning in humans. Typically, theories of practical reasoning make use of a folk psychology, and behaviour is modelled using “mentalistic” attributes such as beliefs, desires, and intentions. Probably the best known type of practical reasoning is the belief-desire-intention (BDI) model. BDI agents are characterised by a “mental state” with the three components belief, desire, and intention. Intuitively, belief corresponds to the information the agent has about its environment. Desires represent “options” available to the agent and intentions are the goals the agent has chosen. The philosophical foundations of the BDI model can be found in Bratman’s account of the role that intentions play in human practical reasoning (for more details see [Bra87]).

1.2.3 Agents and Logic

Daniel Dennett [Den78, Den87] recognised that, depending on the complexity of a system, different ways to describe it were appropriate, and he identified three different stances — the technical stance, the design stance, and the intentional stance.

While it makes sense to talk about a light switch in technical terms, it makes
less sense when talking about a digital or even mechanical watch. In those instances, talking about design rather than technical subtleties makes its working much easier to grasp. If we look at a computer, which is another step up the complexity ladder, talking about design does not clarify matters any more — we need an intentional stance to describe the system. However, the different stances are “backwards compatible”; we can talk about a light switch (a) in terms of electrical currents being closed and disrupted (technical stance), (b) as a system built to show the time (design stance), or (c), using the intentional stance,

“[…] as a (very cooperative) agent with the capability of transmitting current at will, who invariably transmits current when it believes that we want it and not otherwise; flicking the switch is simply our way of communicating our desires.” [Sho90]

While the intentional stance sounds a bit odd when talking about a light switch, it arguably describes the switch more concisely than the technical stance, while still describing its (relevant) aspects. Note that, while the intentional stance is a necessary condition for agency, it is certainly not a sufficient one. Since agents are highly complex systems, it seems that in order to keep the description understandable and manageable, we should use an intentional system. However, while notions like belief and intention have a perfectly clear meaning in “folk psychology”, they are quite hard to define in a logical framework.

As always in logic, two sides of the coin have to be addressed: Syntax, and Semantics. For syntax, there are two fundamental approaches to formulating intentional systems, namely by using a modal language which contains non-truth-functional modal operators which are applied to formulae; or by using a meta-language, typically a many-sorted first-order language containing terms which denote formulae of some other object language. On the semantic side we also have two basic approaches. The best known, and probably most widely used, is
a possible worlds semantics, where an agent’s beliefs, knowledge, intentions etc. are characterised as a set of possible worlds, with an accessibility relation holding between them. We refer the reader to Section 2.1 for a more detailed introduction to modal languages and possible worlds semantics. The most common alternative to the possible worlds approach is to use a sentential, or interpreted symbolic structures approach. In this scheme, beliefs are symbolic formulae which are explicitly represented in the agent’s belief structure.

Logics based on the BDI model are arguably the most widely adopted logics in agent technology (for a by no means complete but diverse overview see [RG91, RG95, FMP95, KM98]). Originally proposed by Rao and Georgeff [RG91], it is based on Bratman’s theory of intentions [Bra87]. Contrary to most other philosophical theories of practical reasoning, Bratman argues that intentions, rather than being reducible to beliefs and desires, are on par with those, and play an important role in practical reasoning. Accordingly, Rao and Georgeff’s formalism features three distinct modalities for belief, desire and intention.

Another logical framework for reasoning about agents is the KARO-framework [vdHvLM93a, vL96], that has its roots in Moore’s system of knowledge and action [Moo80]. KARO stands for knowledge, ability, result, and opportunity, and is based on a somewhat different theory of practical reasoning, where the notions of knowledge and action are central.

1.2.4 The Role of Agents in Modern Computer Science

As discussed, there is no clear definition of what an agent system comprises. Let us stress again that we define agent systems on an methodological rather than an implementation level — it is very well possible to develop an agent system using agent theories, and implement it in some object oriented language, or in BASIC, for that matter. In fact, many supporters of object oriented design fail to see the
difference between object and agent design.

Nevertheless, nowadays applications in different domains use agent technology. They can roughly be divided into several categories (based on [JSW98]):

1. **Industrial Applications**: Industrial applications were among the first Agent based systems, and range from Manufacturing Control over Process Control and Telecommunications to Air Traffic Control [RG95] and Transportation Systems. What is common to these areas is that they can quite naturally be described using agents, and that the communication between those agents is usually limited and well defined (machines in factories have only a few sensors, and their reading seldom requires “interpretation”). In Manufacturing and Process Control for example, there are different machines and process-steps connected by the way they are set up physically. In Air Traffic Control, each airplane entering the airspace of some airport is represented by an agent, as are the different traffic control systems.

2. **Commercial Applications**: While industrial applications are highly complex and specialised systems, commercial applications are more geared towards the mass market. They deal mainly with Information Management — examples are *information filtering* systems that filter relevant data out of email and newsgroups, and *information gathering* systems, which retrieve relevant data from different sources, the most prominent being the World Wide Web. These applications are maybe the best known examples of agent technology (outside the research community). Other applications can be found in e-commerce and Business Process Management. The beginnings of agents that eventually will sell and buy things autonomously can be found in systems that search for cheap flights.

3. **Entertainment Applications**: Although few people consider games to be “serious” applications, they often are taxing the frontier of computer
hard- and software. Agents have obvious roles in computer games — they can take over the role of another player, give rise to charismatic entities in role playing games, support the player by taking over certain (mostly not very interesting or repetitive) tasks, or even simulate pets (e.g. Creatures [GC98]). Another application is interactive Theater and Cinema. Here agents are artificial entities that interact with the user in a “believable” manner. They can be found as charismatic guides on web sites\(^2\), as well as real estate salesmen and women [Cas01].

4. **Ubiquitous Computing:** A relatively new area in which agent based systems are used is ubiquitous computing, also known as pervasive computing, or ambient intelligence. It is characterised by the use of many (small) devices that have computational resources (often additionally to their original function), which work together in ad-hock networks, providing the user with context aware services, allowing the user to access data from mobile devices, optimally hiding the underlying complexity.

The above list being by no means complete, but it suffices to illustrate that agent technology can and is used in a wide range of areas. For more information on above examples we refer to [JSW98].

\(^2\)One example is [http://www.artificial-life.com](http://www.artificial-life.com)
Part I

Single Agents
Chapter 2

Background — Logic and Agents

2.1 Modal Logics

Before we can talk about logics being used to describe agents, we need to introduce the basic logic that many of them use: modal logics [BdRV01].

Modal logic in the 20th century was mainly used by philosophers looking for formal ways to describe necessity. The first to discuss modal logic, however was Aristotle (384BC – 322BC). In *de interpretatione* he noticed that necessity not only implied possibility (and not vice versa) but also that the notions of necessity and possibility were inter-definable. The proposition “$p$ is possible” may be defined as “not-$p$ is not necessary”. He also pointed out that knowing that $p$ and $q$ are possible, does not mean that $p \land q$ are possible. For example, if $p$ again stands for “it is raining”, and $q$ for “it is not raining”, then it is possible that it is raining or that it is not raining, but it is certainly not possible that it is and is not raining.\(^1\)

In the time after Aristotle, some other philosophers worked on modal logic (although they did not call it that), but the interest was generally low. In the 20th

\(^1\)Of course, this example does not apply to the lovely city of Liverpool, where it is certainly possible that it rains and not-rains at the same time.
century, Lewis [Lew12] constructed, in his search for an axiom system for strict implication, several systems which he named S1 to S5. Two of those systems, S4 and S5, are still in use today.

It was not until Kripke [Kri63] however that modal logic had its breakthrough. Based on the work of Hintikka [Hin62], Kripke developed the possible world semantics, which is the most widely used semantics for modal logics today. He introduced a domain of possible worlds, and an accessibility relation between those worlds. Necessarily \( p \) became “\( p \) is true in all accessible worlds” (see Figure 2.1). Logical operators are typically used to capture navigation through these worlds. \( \Box \varphi \) stands for “it is necessarily true that \( \varphi \)”, and \( \Diamond \) denotes “it is possibly true that \( \varphi \)”. Note that \( \Box \varphi \) also holds in a world with no successor-worlds.

Having an intuitive way to interpret modal formulae, modal logic was used to model different philosophical concepts. While those logics all had a rather similar syntax, their interpretation was different. Philosophers built deontic (“ought and must”), epistemic (knowledge and belief), and temporal (sometimes and always) logics. Common to them is that they deploy the modal operators \( \Box \) and \( \Diamond \), and that they are interpreted in a Kripke-style semantics. Also, the two modalities \( \Box \) and \( \Diamond \) are interrelated. Something is necessary if it is not possible that it is not the case, or \( \Box \varphi \equiv \neg \Diamond \neg \varphi \).
The differences between above mentioned systems lay thus not so much in the syntax but rather in restrictions on the relations between the different worlds. In temporal logic, for example, one can view time as being a straight line of distinct time points, or like a tree that is rooted in the present and branches towards the future. In the first case, we would restrict the relation between the different worlds to be transitive and serial (and possibly reflexive), whereas in the latter case, we would use a transitive relation only.

In epistemic logic, the logic of knowledge, philosophers argue(d) about the best / most closely related / most reasonable way to represent knowledge [FHMV95]. Most people would agree to the following statement: “if I know $\varphi$, then $\varphi$ must be true”. Less people would consent to: “if I know $\varphi$, then I also know that I know $\varphi$”. Still fewer people would say that “If I do not know $\varphi$, then I know that I do not know $\varphi$” is valid. In fact, humans most certainly do not know what they do not know.

However, modal logics, and in particular the logics $\text{KD45}$ and $\text{S5}$, have been adopted in logic based specifications of rational agents, representing agent beliefs and knowledge, respectively.

### 2.2 BDI

In the beginning of the 1980s, researchers focussed on “rational” agents, based on one of the most influential philosophical theories, namely Bratman’s theory of practical reasoning [Bra87]. Typically, such theories of practical reasoning make use of a folk psychology, and behaviour is modelled using attributes such as beliefs, desires, and intentions. Probably the best known type of practical reasoning is the belief-desire-intention (BDI) model. BDI agents are characterised by a “mental state” with the three components belief, desire, and intention. Intuitively, belief corresponds to the information the agent has about its environment. Desires
represent “options” available to the agent and intentions are the goals the agent has chosen.

While, for example, Cohen and Levesque [CL90] tried to formalise reasoning using just beliefs and goals within a temporal structure, Rao and Georgeff [RG91] followed Bratman who argued that intentions were not reducible. They created a framework in which belief, intentions, and desires were treated as first class citizens.

Their formalisation of BDI concentrates on two elements. For one, they use first order branching time temporal logic to encode the choices that an agent has at any given moment in time. Secondly, they use modal operators for each of the three dimensions of belief, desire, and intention. The modalities are interpreted in a Kripke model. Worlds within the model are time trees, so at each world, a whole temporal structure representing all the agent’s (and within a multi-agent setting possibly other agent’s) choices exists. The relationships between belief, intentions, and desires can be described in different ways, using different axioms. One such axiom would, for example, state that goals are also to be believed. The same goes for the relation between intentions and beliefs. More interestingly, perhaps, Rao and Georgeff introduce an axiom called “no indefinite referral” does not allow the agent to postpone an intention indefinitely.

2.2.1 Syntax and Semantics

We will now present the formal theory of Rao and Georgeff’s BDI logic [RG91]. As mentioned above, it is based on branching time temporal logic, similar to CTL*[ES89]. However, they extend it to use first-order logic, and add modalities.

Like CTL*, we define two types of formulae: state formulae, which are evaluated with a certain world and moment in time, and path formulae, which are evaluated along a certain path in a given world. State formulae are
2.2 BDI

• any first order formula using the usual connectives and quantifiers;
• special predicates succeeds(e), fails(e), succeeded(e), failed(e), and done(e);
• special predicates \( \text{BEL}(\varphi), \text{GOAL}(\varphi), \text{INTEND}(\varphi) \); and
• \( \text{optional}(\varphi) \) where \( \varphi \) is a path formula

Path formulae are defined as follows:

• every state formula is a path formula; and
• if \( \varphi \) and \( \psi \) are path formulae, so are \( \neg \varphi, \varphi \land \psi, \varphi U \psi, \lozenge \varphi, \text{ and } \Box \varphi. \)

In order to interpret formulae, Rao and Georgeff define three distinct elements, one for the interpretation of state and path formulae, a possible world semantics for beliefs etc, and a semantics to interpret events.

We start out with an interpretation \( M = (W, E, T, \prec, U, B, D, I, \Phi) \), where \( W \) is a set of worlds, \( E \) a set of primitive event types, \( T \) a set of time points, \( \prec \) is a total, transitive, and a backward-linear relation on time points, \( U \) is a universe of discourse, and \( \Phi \) is a mapping from first-order entities to elements of \( U \) for any given world and time point. \( B, D \) and \( I \) are mappings of the type \( W \times T \times W \).

Worlds themselves are time trees, defined as \( \langle T_w, A_w, S_w, F_w \rangle \), where \( T_w \subseteq T \) is a set of time points in a world \( w \), and \( A_w \) is a subset of the relation \( \prec \), restricted to time points in \( T_w \).

Time itself is represented relative to events. The mappings \( S : T_w \times T_w \mapsto E \) and \( F : T_w \times T_w \mapsto E \) connect adjacent time points by events, representing success and failure of some event at some point in time. Events are thus defined so that not doing an action is not equal to trying an action and not succeeding, as failure would be interpreted by traversing to a different time point.

We will, at this point, omit the precise definition for the semantic mapping function \( \models \), and instead discuss some of the axioms that Rao and Georgeff defined.

Belief is modelled using a KD45 modal logic, while modal axioms \( D \) and \( K \) apply to goals and intentions, so goals and intentions have to be consistent and
be closed under implication. Necessitation holds for all three modalities. Further axioms define the relationship between them. Some of them are goal-belief compatibility (any goal is believed), goal-intention compatibility (any goal is intended), intention-to-action (intentions to do primitive actions are actually executed — whether they are successful is another story). Furthermore, intentions are believed to be intentions, as are goals; also, the agent is aware of primitive actions that are happening.

Other axioms can influence how agents behave. For example, blindly committed agents will retain their intentions until they believe them to be accomplished. Other possibilities include single-minded agents, where intentions are kept only as long as they are indeed still believed to be possible.

2.2.2 Implementation

While BDI describes a set of theories about practical reasoning which are generally based on some sort of modal logics, it is also used as a basis for (multi-)agent architectures. In this section, we will focus on the latter. There are several implementations that are based the BDI approach, for example PRS [GLB85], Agent0 [Sho90], dMARS [Kin93, dKLW98], and AgentSpeak(L) [Rao96].

Agent0 was a fairly simple language based on an explicit notion of time. It incorporated modalities for belief, commitment, and abilities, and was implemented in LISP. Its importance nowadays lies mainly in it being the first implementation of a rational agent theory.

While Rao and Georgeff were clearly aiming at providing a specification language for agents rather than a way to implement agents, Rao did create an implementation called AgentSpeak(L) [Rao96]. A more recent implementation, called JASON [BH+04], implements not only AgentSpeak but several extensions [BBJ+02, MVB03, AMHB04] have been proposed and implemented as well.
An AgentSpeak(L) agent is created by the specification of a set of base beliefs and a set of plans. A belief atom is simply a first-order predicate in the usual notation, and belief atoms or their negations are termed belief literals. An initial set of beliefs is just a collection of ground belief atoms. AgentSpeak(L) distinguishes two types of goals: achievement goals and test goals. Achievement goals state that the agent wants to achieve a state of the world where the associated predicate is true. (In practice, these initiate the execution of subplans.) A test goal returns a unifier for the associated predicate with one of the agent’s beliefs; they fail otherwise. A triggering event defines which events may initiate the execution of a plan. An event can be internal, when a subgoal needs to be achieved, or external, when generated from belief updates as a result of perceiving the environment. Plans refer to the basic actions that an agent is able to perform on its environment. Such actions are also defined as first-order predicates, but with special predicate symbols (called action symbols) used to distinguish them from other predicates. A plan is formed by a triggering event (denoting the purpose for that plan), followed by a conjunction of belief literals representing a context. The context must be a logical consequence of that agent’s current beliefs for the plan to be applicable. The remainder of the plan is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan, if applicable, is chosen for execution.

While Rao never provided a formal semantics, a abstract interpreter in Z [dL98] and later a Plotkin-Style Operational Semantics for AgentSpeak have been provided [MB03].

Another example of a BDI architecture is Interrap [Mül97]. It is a hybrid architecture consisting of two vertical layers: one containing layers of knowledge bases, the other containing various control components that interact with the
knowledge bases at their level. The lowest control component is the world interface that manages the interactions between the agent and its environment. Above the world interface there is the behaviour-based component, whose task it is to model the basic reactive capabilities of the agent. Above this component there is a local planning component able to generate single-agent plans in response to requests from the behaviour-based component. On top of the control layer there is a social planning component. The latter is able to satisfy the goals of several agents by generating their joint plans. A formal foundation of the Interrap architecture is presented in [FJ97].

2.3 Default Logics

Another approach to agent programming that extends the BDI concepts with obligations is the so-called BOID architecture (Belief, Obligations, Intentions, Desires) [BDH+01, BDH+02]. The BOID architecture is an extension of BDP logic [Tho00] which again is rooted in Reiter’s default logic [Rei80]. BDP logic is based on conflict resolution for conditional beliefs and desires. BOID logic extends this with conditional obligations and intentions.

Its motivation is to engineer agents that not only can reason about their internal states defined in terms of beliefs, desires etc., but that also can deal with social norms and obligations. Norms (see e.g. [Dig99]) are clearly a concept that can and, maybe should, play an important role when designing high-level theories about agents, because agents not only are meant to operate in different environments that might impose different rules; many of those rules are not defined in clear cut “laws” but rather obligations and norms that an agent should, but not necessarily has to, adhere to. Furthermore, once we allow for the existence of rule sets that are external to the goals and wishes of an agent, we are bound to encounter situations where an agent’s desires and obligations conflict, or where
an agent needs to choose between following norms or achieving some goal. The BOID architecture provides a framework in which those conflicts can be resolved within a rigid logical system based on default logic.

Default Logic defines extensions in terms of so called default theories. Essentially, given some theory, default rules are applied repeatedly until some fix-point is reached.

BOID logic is based on similar principles; instead of using modalities to model each attitude, they are defined by sets of propositional logical formulae representing the agent’s beliefs, obligations, intentions, and desires. Default rules for each set are iteratively applied to the current theory. Whenever a conflict arises, a preference relation over the default rules is used to solve it. Different preference relations define different types of agents. Broersen et al. [BDH+01] define four different such types, three of which are a subset of the first, namely: realistic, where beliefs override all other components; simple-minded agents (BIDO, BIOD) prefer intentions to desires and obligations; selfish agents (BDIO, BDOI) let their desires overrule obligations; finally, social agents (BIOD, BOID, BODI) value obligations higher than their own desires. While there are more types possible, the given ones cover “rational” agents, in that belief always is preferred to other attitudes.

2.3.1 Syntax and Semantics

The BOID logic is given as an extension to Reiter’s characterisation of extensions (see [Rei80]). We start out with a propositional language $L$ and a set $S$ of ordered pairs of $L$ called rules. We write elements of $S$ as $\alpha \rightarrow \omega$. An agent type is a set of functions $\rho : S \times \mathbb{N}$ from $S$ to the set of natural numbers.

Agent types are generally expressed in terms of constraints. For example, given a simple language that only contains beliefs and desires, a realistic agent
type is expressed by the constraint that for all \( r_b \in B \) and \( r_d \in D \), we have \( \rho(r_b) < \rho(r_d) \).

Given some agent type, we define the calculation scheme for BOID extensions as follows:

Given a propositional language \( L \), we define a BOID theory as a tuple \( \Delta = \langle W, B, O, I, D \rangle \), where \( W \) is a (sub-) set of \( L \), and \( B, O, I, D \) are sets of ordered pairs \( \alpha \leftarrow w \) of \( L \). Let \( \rho \) be a function that assigns to each rule in \( B \cup O \cup I \cup D \) a unique integer, and \( S \) be a subset of \( L \). Now, we define

\[
\rho_{\text{min}}(\text{BOID}, S) = \min\{\rho(\alpha \leftarrow w) | B \cup O \cup I \cup D, \alpha \in S, \neg w \notin S\}
\]

\[
\min(\text{BOID}, S) = \text{w.s.t. } \alpha \leftarrow w \in B \cup O \cup I \cup D, \rho(\alpha \leftarrow w) = \rho_{\text{min}}(\text{BOID}, S)
\]

We now iteratively create extensions, \( E \). First, we define \( E_0 = W \) as the basis. We then define for \( i \geq 0 \):

\[
E_{i+1} = \text{Th}_L(E_i \cup \{\min(\text{BOID}, S)\})
\]

if such a minimal element exists, and \( E_{i+1} = E_i \) otherwise. Finally, we can define an extension \( E \subseteq L \) to be an extension for \( \Delta \) of agent type \( A \) if and only if \( \exists \rho \in A \) s.t. \( E = \bigcup_{i=0}^{\infty} E_i \).

### 2.3.2 Implementation

There exist two implementations of the BOID architecture [BDH+01]. Both adopt a similar control loop, but treat the way defaults are applied differently. The control loop, shown in Figure 2.2, works as follows. At each iteration, the agent uses observations (which are non-refutable, as they reflect the true state of the world), and creates an extension based on the application of default rules to those observations.

The first implementation uses a single-extension BOID. Essentially, it is a
simple prioritised production system that iterates through the following loop: First, it finds all rules that are applicable to the current extension. Then it selects the rule with the lowest $\rho$ value and applies the rule, which results in a new extension set. This is repeated until either no more rules are applicable, or until a fixed-point has been reached.

The second implementation approximates a multi-extension BOID. It does this by, rather than using $\rho$ to order rules, using the four-letter order to prioritise them. Extensions are built by iteratively applying rules of different components. For example, a BOID agent type would first apply its belief component, followed by one intention rule, and feeding the result back into the belief component. Then, D rules are applied, which are fed into B and I components again. Conflicts within one component are solved by simple ordering — Prolog processes rules in a top-down fashion. From this it follows as well that multiple extensions can be created using Prolog’s backtracking mechanism.

2.4 Situation Calculus

The situation calculus is a first order language with some second order features that was originally developed to formalise dynamic aspects of the world, including
concepts of ability and belief [LPR98, MH69, PR99]. The logic itself is rather versatile, and has been expanded to include concurrency [GLL98, GLL99], time [PR93], non-monotonic reasoning [Bak91] events, and more [Lak99, Lin95]. We will outline the basic constructs, and present a programming language based on the situation calculus.

2.4.1 Syntax and Semantics

The language draws on three disjoint sets, denoting actions ($L_{ac}$), situations $L_s$, and object ($L_o$) for everything else. Its alphabet consists of the following.

- Countably many variable-symbols for each sort, especially $\{s_1, \ldots\}$ for situations and $\{a_1, \ldots\}$ for actions. Since the language essentially is second order, variables for predicates of all arities are also included.
- A special situation constant $S_0 \in L_s$, denoting the initial situation.
- A binary function symbol $do : L_a \mapsto (L_a \mapsto L_s)$. The idea is that $do(a, s)$ denotes the successor situation resulting from performing $a$ in situation $s$.
- A binary function symbol $Poss : L_a \mapsto L_s \mapsto \{true, false\}$. The boolean value is intended to denote whether a given action is possible in some situation.
- A proper ordering $\sqsubseteq$ on situations.
- Countably many predicate and function symbols (yielding an object) of arity $n \in \mathbb{N}$, with arguments $(L_a \cup L_o)^n$. The intended interpretation are situation independent relations.
- Countably many function symbols of finite arity yielding actions. These we call action functions.
- Countably many predicate symbols with arguments $(L_a \cup L_o)^n \times L_s$, called relational fluents. Their value is dependent on the situation.
- Countably many function symbols with arguments $(L_a \cup L_o)^n \times L_s$, yielding $L_a \cup L_o$, called functional fluents.
Generally, a particular domain will be expressed by the union of the sets of axioms:

- Fundamental axioms $\Sigma$, stating (a) that situations have a unique name, (b) an induction axiom on situations, and (c) finally axioms describing the $\sqsubseteq$ relation.
- Action precondition axioms ($Poss$), one for each action.
- Successor state axioms, for each fluent.
- Axioms to describe primitive actions.
- Axioms describing the initial situation $S_0$.

Formulae in Situation Calculus are usually interpreted over a set of axioms. The main techniques used here are regression and $\Sigma$-elimination. Regression is based on the idea that, in every situation, it is possible to trace back the sequence of actions to the initial situation. A situation can be seen as a possible world history, that is a sequence of actions that led to the current state of affairs. Note that, initially, situations represent a snapshot of the world, rather than a history.

One problem of using logic programming of any sort is the frame problem. Generally speaking, the frame problem talks about the fact that, in a reasonably complex environment, every action changes only a bit, but all possible combinations of properties and outcomes need to be considered. For Example, walking through a room does not change the colour of the wall — common sense never even considers that, but a naive logical calculus does.

In situation calculus, this is solved by allowing for the successor state axiom, which states (for each fluent) all the conditions under which it can change. This solution rests on a completeness assumption — that we are able to extensively describe the conditions under which a fluent changes — which in turn does not allow for state constraints.\footnote{There are extensions of situation calculus that deal with causality though. See e.g. [Lin95].}
2.4.2 Implementation

Golog is a high level specification language, based on the situation calculus, extending it with control structures like sequence, test actions, non-deterministic choice (of actions as well as programs), and most importantly procedures. An evaluation of a Golog program basically corresponds to proving the theorem expressed by a program against some background theory (expressed by its axioms). As a by-product, variables are instantiated and provide an execution trace. In that it is rather similar to Prolog. In fact, a Golog interpreter can easily be written in Prolog. Procedures in Golog really are macros. A given program is expanded to be one formula that is evaluated.

As with situation calculus, the basic Golog programming language can be extended considerably. One of the most prominent extensions is ConGolog [GLL98, GLL99], a concurrent version. It incorporates priorities of threads, as well as interrupts. One must notice that the concurrency described (e.g. [GLL98]) is not true concurrency but rather an interleaving of processes. (Of course, on a one-processor machine this is always the case, but “truly” concurrent programs let the operating system take care of the actual interleaving, setting only their priority.) This means that Golog can be programmed to either exhibit coarse or fine interleaving. In extremo, it is possible to interleave whole procedures / programs rather than steps. Even when interleaving step-wise, problems arise the moment we execute actions of a long duration.

Interrupts, together with prioritising processes, can be a natural way of specifying certain tasks, and getting reactive rather than pre-set behaviour.
2.5 KARO

The foundations of KARO originate in the work of Moore [Moo80, Moo84] on knowledge and action. Moore realised that until then, the methods to deal with agent programming were not doing much more than representing all available data and letting a search algorithm loose on it. Moore’s approach was more subtle, trying to capture the notion of knowledge and action as two distinct concepts.

Van der Hoek, van Linder, and Meyer proposed a logic of capabilities in 1993 that built on Moore’s theory, but extended it in several ways [vdHvLM93a]. For one, they simplified the syntax of Moore’s language. More importantly, they treated abilities as first class citizens, rather than mere prerequisites for executing actions. This allowed them to remove abilities as prerequisites for events, which should be independent of both the knowledge and abilities of agents, and treat abilities instead on the same level as events. In the resulting language, agents are able to reason about the result of actions without necessarily being able to actually perform those actions. The term “abilities” is to be understood in a broad sense, that is, it describes not only physical, but also mental and even moral capacities of an agent. There are more important notions connected with events than just abilities. For example, the mere ability to perform actions does not suffice, as the actual performance of these actions must return results.

Another notion connected to events is opportunity, which can be best described as “circumstantial possibility”. Often opportunity is nothing more than the absence of circumstances that would prevent performance. In KARO, opportunity is implemented by means of an accessibility relation.

Over the years, the KARO framework was extended numerous times, augmenting the core language with formalisations for commitments, wishes, and goals [vdHvLM93a, vL96, vLMvdH97, vdHvLM97, vLvdHM98, MvdHvL99, vdHMvS99]. Also, proof methods were researched, as was the influence of (non-)
determinism and more [vdHvLM93b, JJdBvE+01, HDS+01]. However, the core
language of knowledge, ability, result and opportunity always stayed the same.

While opportunities were modelled using accessibility relations, abilities were
formalised using a so-called capability function. Not only did this pose technical
difficulties\(^3\), but the approach also suffered from strange properties regarding the
ability of the agent to perform sequentially composed actions. Depending on
whether actions were interpreted using an “optimistic” or “pessimistic” stance
[vLvdHM98, vL96], agents would, in a counterfactual state, either be able to do
everything, or nothing.

In van der Hoek’s et. al. new proposal [vdHMvS99], both abilities and oppor-
tunities are considered to be instances of the potential of an agent. Instead of
using a capability function or an accessibility relation to model ability and op-
portunity respectively, potentials are represented using standard dynamic logic.

Not only does this render the system less complicated, it also avoids the
above mentioned difficulties with sequential actions, because in the new language
abilities and opportunities are not interrelated any more.

2.5.1 Syntax and Semantics

KARO Syntax

Let \( \Pi \) be a set of propositional atoms, \( \text{Ag} \subseteq \mathbb{N} \) a set of agents, \( \text{At} \) a set of
atomic actions, and \( \text{Lb} \) a set of potentials. Henceforth, unless otherwise stated,
\( \text{Lb} = \{0, A, E\} \), where 0 stands for opportunity, A for ability, and E for of executing
an action. Then, the ordered tuple \( \Sigma_K = (\Pi, \text{Ag}, \text{At}, \text{Lb}) \) is a KARO-signature.

The set of well-formed formulae \( \mathcal{L}_K(\Sigma_K) \) KARO logic over a signature \( \Sigma_K \) is
inductively defined as follows:

\(^3\)For example, an axiomatisation of KARO used rules with infinitely many premises, called
\( \omega \)-rules [vdHvLM93a, vL96].
2.5 Karo

\[ \Pi \subseteq L_K(\Sigma_K), \]
\[ \text{if } \varphi_1, \varphi_2 \in L_K(\Sigma_K), \text{ then } \neg \varphi_1, \varphi_1 \land \varphi_2 \in L_K(\Sigma_K); \]
\[ \text{if } \varphi \in L_K(\Sigma_K), i \in \text{Ag}, \text{ then } K_i \varphi \in L_K(\Sigma_K); \text{ and} \]
\[ \text{if } X \in \text{Lb}, i \in \text{Ag}, \alpha \in \text{Ac}(\Sigma_K), \text{ and } \varphi \in L_K(\Sigma_K) \]
\[ \text{then } (i, X, \alpha) \varphi \in L_K(\Sigma_K). \]

Here the set \( \text{Ac}(\Sigma_K) \) of action formulae is inductively defined as follows:

\[ \text{At} \subseteq \text{Ac}(\Sigma_K); \]
\[ \text{if } \varphi \in L_K(\Sigma_K), \text{ then } \varphi ? \in \text{Ac}(\Sigma_K); \text{ and} \]
\[ \text{if } \alpha_1, \alpha_2 \in \text{Ac}(\Sigma_K), \text{ then } (\alpha_1; \alpha_2), (\alpha_1 + \alpha_2), (\alpha_1^*) \in \text{Ac}(\Sigma_K). \]

As in \( \mathcal{PDL} \), \( (\alpha_1; \alpha_2) \) denotes sequential composition, \( (\alpha_1 + \alpha_2) \) choice, and \( (\alpha_1^*) \) repetition. We consider \( \varphi \lor \psi, \varphi \rightarrow \psi, \text{ and } [i, X, \alpha] \varphi \) as abbreviations for \( \neg(\neg \varphi \land \neg \psi), \neg(\varphi \land \neg \psi), \text{ and } \neg(i, X, \alpha) \neg \varphi, \) respectively.

\( (i, X, \alpha) \varphi \) is read as: \( \varphi \) is a possible result of \( i \) executing \( \alpha \), having the \( X \)-potential to do so (except for \( X=E \), which stands for the actual execution). Note that this does not imply that \( i \) has other potentials. In the given setting, an agent can very well have the opportunity to perform \( \alpha \) without having the respective ability, and vice versa.

Karo formulae are interpreted in a Kripke-style semantics.

**Karo Interpretation**

Let \( \Sigma_K = (\Pi, \text{Ag}, \text{At}, \text{Lb}) \) be a Karo-signature. A Karo-interpretation \( \mathcal{M} \) over \( \Sigma_K \) is an ordered tuple \( (W, E, D, \pi) \), where

- \( W \) is a set of states.
- \( E : \text{Ag} \mapsto 2^{W \times W} \) maps each agent to an equivalence relation over \( W \).
- \( D \) is a set of relations, which contains, for all \( X \in \text{Lb} \), a relation \( p_X : \text{Ag} \mapsto (\text{At} \mapsto 2^{W \times W}) \), which maps every agent and every (atomic) action to a
binary relation over $W$. Note that the relations are only defined for atomic actions.

- $\pi : \Pi \mapsto 2^W$ is an evaluation function that assigns to each atom $p \in \Pi$ a set of states in which $p$ is true.

$E$ takes care of the epistemic part of the logic. Given a model $M$ and a state $s$, $K_i \varphi$ is true in $s$ if, and only if, $\varphi$ is true in all states reachable from $s$ via the $E(i)$-relation. Since the relation is required to be an equivalence relation, $K_i$ is a $S5$ modality. In addition to the epistemic modalities $K_i$, we have the potential modalities $\langle i, X, a \rangle$, for each potential, agent and atomic action. That means that for $a \in At$ every modal operator $\langle i, X, a \rangle$ is interpreted by means of the relation $p_X(i)(a)$. However, there is no relation for composite actions. Those will be interpreted using a lifting function $\hat{p}_X$, conforming to Goldblatt [Gol92], as follows.

Let $M^K = (W, E, D, \pi)$ be a KARO model over a Signature $\Sigma_K = (\Pi, Ag, At, Lb)$, $\alpha_1, \alpha_2 \in Ac(\Sigma_K)$ be action formulae, and $s, u, t \in W$ be worlds of the model. Then $\hat{p}(\alpha)(i)(\alpha)$ is defined as

$$\hat{p}_X(i)(\alpha)(a) = p_X(i)(a) \in D$$

$$\hat{p}_X(i)(\varphi ?) = \{(s, s) \mid s \in S, M, s \models \varphi\}$$

$$\hat{p}_X(i)(\alpha_1; \alpha_2) = \hat{p}_X(i)(\alpha_1) \circ \hat{p}_X(i)(\alpha_2)$$

$$= \{(s, t) \mid \exists u : (s, u) \in \hat{p}_X(i)(\alpha_1) \land (u, t) \in \hat{p}_X(i)(\alpha_2)\}$$

$$\hat{p}_X(i)(\alpha + \alpha_2) = \hat{p}_X(i)(\alpha_1) \cup \hat{p}_X(i)(\alpha_2)$$

$$\hat{p}_X(i)(\alpha^*) = (\hat{p}_X(i)(\alpha))^*$$

$$= \{(s, t) \mid \exists k \exists s_0, s_1, \ldots, s_k : (s_0 = s) \land (s_k = t)$$

$$\land (\forall 0 \leq i \leq k : (s_{i-1}, s_i) \in \hat{p}_X(i)(\alpha))\}$$

Here, $\circ$ denotes the composition of two binary relations, and $^*$ stands for the reflexive transitive closure of a relation.

Finally we present the semantics of well-formed KARO formulae.
Karo Semantics

Let $\Sigma_K = (\Pi, Ag, At, Lb)$ be a Karo-signature. We inductively define a relation $|$ between Karo-interpretations over $\Sigma_K$, states, and Karo formulae over $\Sigma_K$ as follows.

- $\mathcal{M}, s \models p$ if and only if $s \in \pi(p)$
- $\mathcal{M}, s \models \neg \varphi$ if and only if $(\mathcal{M}, s \not\models \varphi)$
- $\mathcal{M}, s \models \varphi \land \psi$ if and only if $(\mathcal{M}, s \models \varphi)$ and $(\mathcal{M}, s \models \psi)$
- $\mathcal{M}, s \models K_i \varphi$ if and only if for every $s'$ such that $(s, s') \in E(i)$, $\mathcal{M}, s' \models \varphi$
- $\mathcal{M}, s \models \langle i, x, \alpha \rangle \varphi$ if and only if there exists $s'$ such that $(s, s') \in p_X(i)(\alpha)$ and $\mathcal{M}, s' \models \varphi$

If $\mathcal{M}, s \models \varphi$ then we say $\varphi$ is true or holds at a world $s$ in $\mathcal{M}$. If $\varphi$ is a Karo formula over $\Sigma_K$ such that there is an interpretation $\mathcal{M}$ over $\Sigma_K$ and a world $s$ in $\mathcal{M}$ such that $\varphi$ is true at $s$ in $\mathcal{M}$, then $\varphi$ is satisfiable and $\mathcal{M}$ is a model for $\varphi$.

This definition of Karo only covers the core language.

2.5.2 Implementation

To our knowledge there exist no direct implementations of Karo, though several languages borrow features, such as 3APL, Dribble and GOAL, which we describe here.

3APL

Hindriks [HBdHM99, Hin01] describes 3APL as abstract programming language, or coordination language in that it does not provide any concrete set of actions, nor demands certain knowledge representations. It rather aims at coordinating heterogenous activities and knowledge representation languages of agents. 3APL implements beliefs, observations, actions, goals, communication, and reasoning
rules. In particular, agents’ observations and beliefs can be implemented in 3APL by a subset of first-order predicate logic (prolog-like facts and rules). The actions are implemented as triples consisting of an action name together with pre- and post-conditions. The pre- and post-conditions of actions are belief formulae indicating the condition under which the action can be performed and the effect of the action after it is performed, respectively. The goals that can be implemented in 3APL are procedural or to-do goals, which can be implemented by expressions of an imperative language. These expressions are formed by applying constructs such as sequence, test, conditional choice, and recursion to actions and belief formulae. Communication can be implemented by the pre-defined send-message action. Finally, reasoning rules can be used to implement the generation of goals, the revision of actions that are blocked, the revision of goals that are not achievable, the optimisation of goals, etc.

An 3APL agent consists of four modules with a central interpreter for control, which are beliefs, goals, capabilities, and practical reasoning rules. Triggered by sensing actions, the agent updates and revises its beliefs and goals. Using practical reasoning rules it selects appropriate actions from its capabilities to achieve its goals. Capabilities are of the form \{Pre\}Action\{Post\}. The belief base contains a set of facts. Rules are of the form Goal ← Guard—SubGoals, where subgoals are selected provided the guard condition is satisfied. In this way agents can be programmed that adjust their program to the circumstances.

3APL is a very flexible language in that several different programming languages have been mapped onto it, such as AgentSpeak(L) [HdBvdHM98] and ConGolog [HLL00]. It has also been extended to include declarative goals such as the ones used in GOAL (see next section) [DvRDM03] (in the process subsuming Dribble (see next section) which is based on a propositional language, whereas 3APL has first order features). Fairly new is a Java (and Prolog in Java)-based
3APL platform with which real 3APL based agents can be programmed, and which allows multiple 3APL agents to communicate [Das04].

**Dribble and GOAL**

Another language that is rooted in KARO’s notions of actions and capabilities is GOAL [HdBvdHM01]. Starting out by distinguishing between *goals-to-be* and *goals-to-do*, the authors claim that agent logics generally incorporate the notion of declarative goals, or goals-to-be, whereas implementations define goals as sequences of actions, or goals-to-do. GOAL tries to close this gap by defining declarative goals in its logic. The language is based on two databases. While the belief and goal base are independent, certain constraints exist that keep them consistent with each other. GOAL assumes a (propositional) language as a basis for those databases. Furthermore, agents have capabilities, or (basic) actions that serve to update the belief base (and, by extension as well as directly, the goal base) of the agent.

Dribble [vRvdHM03] combines declarative and procedural features of GOAL and 3APL respectively. The language Dribble thus incorporates beliefs and goals as well as planning features. The idea is that a Dribble agent should be able to select a plan to reach a goal from where it is at a certain point in time. In order to do that, the agent has beliefs, goals and rules to select plans and to create and modify plans. However, it it is based on purely propositional logic, which diminishes its uses somewhat.

**2.6 Logic Programming**

Computational Logic has its roots in the 1950s, with work on the automation of logical deduction, and was fostered in the 1970s [Kow74] as Logic Programming [AP96]. Logic programming defined a new paradigm in programming — that
of declarative programs (as opposed to imperative or object-oriented programs). Essentially, declarative programming focuses on the declarative meaning, leaving the procedural aspects of the execution to the machine. The Prolog language [CKRP73] was and is the most famous declarative language to date. The developments of the formal foundations of logic programming began in the late 1970s (see for example [EK76, Cla78]).

The selection of logic programming as the underlying paradigm for the Japanese Fifth Generation Computer Systems Project [Uch82] led to the rapid proliferation of various logic programming languages. Due to logic programming’s declarative nature, it quickly became a candidate for knowledge representation. While declarative programming never managed to move beyond research and into the mainstream of programming languages, it proved very useful for representing knowledge bases and for reasoning with them. However, its use of the closed-world-assumption, together with the fact that logic programs were essentially monotonic, meaning that conclusions, once reached, will stay true forever, diminished its use as language for (dynamic) agents.

More recently, dynamic logic programs [ALP+98] removed the static nature of logic programs, allowing it to be used in agent programming languages [LAP01a].

We will now give a short introduction to logic programming.

2.6.1 Syntax and Semantics

Logic programs are based on predicate logic. More specifically, a logic program $P$ consists of a set of Horn-clauses which are formulae of the form $L_1 \leftarrow L_2 \land \ldots \land L_n$, where $L_i$ is a positive predicate [Llo87].\(^4\) Clauses which consist of only one positive literal are also called facts.

While other interpretations and models are possible, we will give here the one

\(^4\)Formally, Horn clauses have to have at most one positive literal, so they can also consist of only negative literals.
by Herbrand [Llo87].

A Herbrand universe, \( H \), is defined as follows. Given a set of constants \( C \) and a set \( F \) of functions, \( H \) is defined inductively as the smallest set such that

- any constant in \( C \) is in \( H \); and
- if \( f/n \) is a \( n \)-ary function in \( F \), and \( t_1, \ldots, t_n \) are elements of \( H \), then \( f(t_1, \ldots, t_n) \) is in \( H \).

A Herbrand base is the set of all predicate symbols applied to all possible tuples of elements from the Herbrand universe.

Given a Herbrand universe \( H \) and a program \( P \), we can now define two- and three-valued interpretations of \( P \). In the former case, we define an interpretation function \( I \) of a program \( P \) to be any subset \( S \) of the Herbrand base \( H \) of \( P \). The elements in \( S \) are considered true, and all others are considered false. This is also known as the closed-world-assumption, where everything that is not explicitly known to be true is assumed to be false.

A 3-valued interpretation does not make any such claim. Instead, an interpretation \( I \) of a program \( P \) consists of a set \( T \cup \text{not } F \), where \( T \) and \( F \) are disjoint subsets of \( H \). \( T \) contains all ground atoms that are true in \( I \), while \( F \) contains the ones known to be false. The truth value of the remaining atoms is considered unknown, or undefined.

Alternatively, we define an interpretation function \( I \) which assigns 1, if the argument is true, 0 if it is false, and .5 otherwise. Note that this function is valid for the two- and three-valued case, as in the former there simply is no atom that is not either true or false. Given the function \( I \), we assume a lifting function \( \hat{I} \) which assigns truth values to all elements of the language, rather than just atoms.

Now we can define a model as the interpretation \( \hat{I} \) of a program \( P \) if, and only if, for every ground instance of a program rule \( H \leftarrow B \) we have \( \hat{I}(H \leftarrow B) = 1 \), where \( H \) is a positive and \( B \) a set of negative literals.
Dynamic Logic Programming

Generally, logic programs were dynamic in the sense that their knowledge bases could be updated. An update then consists in changing only those predicates whose values changed, and leaving all others un-changed, following the law of inertia [ALP+98]. This, however, leads to counter-intuitive, and often inconsistent states, because by changing the value of predicates, clauses might yield different values, leading to further predicates that would need changing.

Dynamic Logic Programming [ALP+98] is based on the idea of applying the law of inertia not to the consequences of rules, but rather the rules themselves. This means that when updating programs, rules persist unless they are in conflict with the update.

From this it follows, that, rather than considering just one logic program, sequences of programs are considered. Each state represents, for example, a distinct time period, or a certain set of priorities. Transition rules govern how successor states can be obtained.

Abductive Logic Programming

Logic programming generally implements deductive reasoning. Thus, given a set of rules and some propositions, it establishes whether the rules support the propositions. Abduction on the other hand consists of computing explanations for observations [KKT98], or computing from effects to causes. Generally, using abductive reasoning, multiple possible causes for an effect exist. Also, inferred explanations might turn out to be false in the light of new information.

In logic programming, the proof of a rule fails if a subgoal fails to unify with the head of the clause. Using abductive reasoning, the subgoal is instead considered as a hypothesis, thereby extending the current knowledge base with (consistent) new information.
A set of integrity constraints defines possible abducibles, i.e. predicates that can be abduced. More formally, an abductive logic program is a tuple $\langle P, A, IC \rangle$, where $P$ is a logic program, $A$ a set of abducibles, and $IC$ a set of integrity constraints, each of which contains at least one abducible. Given a program $P$ and a goal $G$, the goal of abduction is to find a set of atoms $\Delta$ which, together with $P$, entail $G$ [LMCT01].

2.6.2 Implementations

Recently, a agent architecture, based on dynamic logic programming, named MINERVA was developed [LAP01a]. It implements agents based on multidimensional dynamic logic programming (MDLP). Rather than viewing dynamic logic programs as a sequence of logic programs, it employs an acyclic graph of logic programs [LAP01b]. Furthermore, agents in MINERVA also contain knowledge about transitions between states, i.e. between logic programs. In [APPP99], a language based on logic programming called LUPS is defined with which such transitions can be specified. It basically defines assertions and retractions of rules based on a guard which is evaluated in the current state.

MINERVA agents consist of a knowledge base which contains the multidimensional logic program, as well as specialised databases for capabilities, goals, intentions, plans, reactions, and internal behaviours, some of which are programmed using LUPS, others using MDLP. The databases are (concurrently) updated by sub-agents performing specialised (sub-)tasks, such as planning, sensing, and communicating with other agents.

For abductive logic programming, an implementation called ALIAS [CLM+00] has been developed, which provides a framework for agents based on abductive logic programs. It is based on a combination of Java and Prolog, and allows agents to communicate over a network. Agents have a statically defined knowledge base.
Working in a group, or *bunch*, they can dynamically build a global knowledge set by applying abductive reasoning and adding such newly acquired knowledge into the global pool, if it is consistent with the local knowledge of the other agents in the bunch, as well as with the knowledge assumed so far. ALIAS uses a communication language called LAILA [LMCT01] to allow the agents to maintain consistency.
Chapter 3

Single METATEM Agents

The preceding chapters provided an overview of both (logic based) specification languages for agents and programming languages for agents. In this chapter, we will present METATEM, the language we use to program our agents. After defining the underlying logic in Section 3.1, we will outline METATEM and its extensions in Section 3.2. After describing some of the details of the implementation we have developed in Section 3.3, we conclude by providing a structural semantics upon which the implementation is based (Section 3.4).

3.1 Temporal Logic

METATEM is based on discrete propositional temporal logic [Gol92] with a finite past and infinite future. Essentially, temporal logic is propositional logic (see e.g. [vBvDKV94]), extended with modalities that are interpreted in a temporal context. Thus, as well as notions of truth and falsity in certain moments of time, there are operators allowing the description of temporal concepts.

More formally, we define the language of temporal logic, $TL$, as follows. Let the well formed formulae of temporal logic ($WFF_{TL}$) be the smallest
set of formulae containing: a signature $\Sigma_{TL} = \langle P \rangle$ of propositional constants (also called its signature), the symbols $\text{true}$, $\text{false}$, and $\text{start}$, and its closure under the connectives $\neg, \land, \lor, \Rightarrow$ as well as the unary temporal operators $\circ$ (next), $\Diamond$ (sometime), and $\Box$, and the binary temporal operators $\circlearrowleft$ (always), $U$ (until), and $W$ (unless).

Temporal formulae are interpreted over a discrete linear model of time with finite past and infinite future, i.e. a sequence of moments in time in which each state provides a classical propositional valuation. We represent this structure by $\mathcal{M} = (\mathbb{N}, \pi_p)$, where $\mathbb{N}$ is the Natural numbers, which we use to represent the sequence of moments $s_0, s_1, \ldots$ and $\pi_p : \mathbb{N} \times P \rightarrow \{\text{True, False}\}$ is a map that assigns to each proposition in $P$ a valuation at each moment in the sequence.

The interpretation for this logic is defined as a pair $\langle \mathcal{M}, i \rangle$, with $\mathcal{M}$ being the model and $i$ the index of the moment in which the formula is to be interpreted.

As usual, we define the semantics of the language via the satisfaction relation $\models$ which holds between interpretations and elements of $WWF_{TL}$ as follows:

$$\langle \mathcal{M}, i \rangle \models \text{start} \quad \text{iff} \quad i = 0$$

$$\langle \mathcal{M}, i \rangle \models \text{true}$$

$$\langle \mathcal{M}, i \rangle \not\models \text{false}$$

$$\langle \mathcal{M}, i \rangle \models p \quad \text{iff} \quad \pi(i, p) = \text{True}, \quad p \in \mathcal{P}.$$  

$$\langle \mathcal{M}, i \rangle \models \neg \varphi \quad \text{iff} \quad \langle \mathcal{M}, i \rangle \not\models \varphi.$$  

$$\langle \mathcal{M}, i \rangle \models \varphi_1 \land \varphi_2 \quad \text{iff} \quad \langle \mathcal{M}, i \rangle \models \varphi_1 \text{ and } \langle \mathcal{M}, i \rangle \models \varphi_2.$$  

$$\langle \mathcal{M}, i \rangle \models \circ \varphi \quad \text{iff} \quad \langle \mathcal{M}, i + 1 \rangle \models \varphi.$$  

$$\langle \mathcal{M}, i \rangle \models \Diamond \varphi \quad \text{iff} \quad \text{there exists } j \in \mathbb{N}, \text{ s.t. } j \geq i \text{ and } \langle \mathcal{M}, j \rangle \models \varphi.$$  

$$\langle \mathcal{M}, i \rangle \models \Box \varphi \quad \text{iff} \quad \text{for all } j \in \mathbb{N}, \text{ if } j \geq i \text{ then } \langle \mathcal{M}, j \rangle \models \varphi.$$  

$$\langle \mathcal{M}, i \rangle \models \varphi U \psi \quad \text{iff} \quad \text{there exists } j \in \mathbb{N} \text{ s.t. } j \geq i \text{ and } \langle \mathcal{M}, j \rangle \models \psi, \text{ and, for all } k \in \mathcal{N}, \text{ if } i \leq k < j \text{ then } \langle \mathcal{M}, k \rangle \models \varphi.$$  

$$\langle \mathcal{M}, i \rangle \models \varphi W \psi \quad \text{iff} \quad (\langle \mathcal{M}, i \rangle \models \varphi U \psi) \text{ or } (\langle \mathcal{M}, i \rangle \models \Box \varphi)$$
Informally, the temporal operators are interpreted as follows (and as depicted in Figure 3.1): the special proposition start is true only at the beginning of time. ‘$\bigcirc \varphi$’ means that $\varphi$ must be true at the next moment in time; ‘$\Diamond \varphi$’ is interpreted as $\varphi$ having to be satisfied at some future moment in time, and ‘$\Box \varphi$’ is true only if $\varphi$ is true at all future states. The binary operators ‘$\varphi U \psi$’ and ‘$\varphi W \psi$’ both state that $\varphi$ holds from the current moment until a moment in the future at which $\psi$ holds. The difference between the two operators is in their relative strength. $\varphi U \psi$ implies that $\psi$ must be satisfied at some point in time, while $\varphi W \psi$ would also be satisfied just if $\varphi$ held at all future points regardless of whether $\psi$ was to be satisfied. $W$ can, and in fact is, defined in terms of $\Box$ and $U$, as the definition of the semantic function $|=\text{ shows}$.

---

**Figure 3.1.** Graphical representation of temporal operators
3.1.1 Temporal Logic and Agency

From the preceding chapter it should be clear that there exist numerous logics that can be, and are, used to describe the behaviour of agents. All of the approaches have their respective advantages. Temporal logic fulfils several of our requirements, namely:

- (Propositional) temporal logic, while more expressive than classical propositional logic, remains tractable.
- Temporal operators allow for concise descriptions of dynamic behaviour.
- The linear discrete model is an intuitive way to describe an execution sequence with a defined beginning and an (potentially) infinite execution.

Temporal logic also is often used to describe processes, especially concurrent and distributed processes [Sti01]. Properties such as liveness and safety can be expressed in very simple terms. For example liveness, the property that ”something good does eventually happen”, can be expressed as $\Box \Diamond \varphi$ (it is always the case that $\varphi$ eventually holds), whereas safety, the property that ”nothing bad ever happens”, can be expressed with the temporal formula $\Box \neg \varphi$ [Lam83]. Furthermore, temporal logic allows to program in a declarative and intuitive manner. For example, we can express “soft goals”, such as $\Diamond \text{re-fuel}$, as opposed to the imperative if low then re-fuel, which (inherently) does not allow for flexible behaviour.

As we will see, we can extend temporal logic with modal operators that cover another important aspect of (rational) agency, namely mentalistic notions that allow agents to employ “practical reasoning”. As we have seen in Chapter 2, mentalistic modalities, and in particular the BDI approach, have been the basis of many different agent logics and languages. In the remainder of this chapter, we will discuss a logic that combines temporal and mentalistic notions.
3.1.2 Executing Temporal Logic

Executing temporal formulae basically means to follow an algorithm that creates a model for the formulae in a constructive fashion.

In [Fis97b], Fisher introduces a translation that, given an arbitrary temporal formula, creates a set of rules called the separated normal form (SNF), which is equivalent to the original formula, but comprises solely of simplified formulae of the form *past or present implies present or future*. Generally, transformation of arbitrary formulae into SNF works by first moving negations in front of propositions, and then replacing complex sub-formulae with propositions whose truth value in all states is linked to the formula that they replace. We refer the reader to above cited paper for a thorough discussion of SNF. For the purposes of this thesis, it is sufficient to realise that arbitrary formulae transformed into SNF have the form

\[ \square \bigwedge_i T_i \]

where \( T_i \), also referred to as *rule*, is of one of the forms in Figure 3.2, where each \( l_b, k_a \) is a literal.

\[
\begin{align*}
\text{start} & \Rightarrow \bigvee_{b=1}^r l_b \quad \text{(an initial rule)} \\
\bigwedge_{a=1}^g k_a & \Rightarrow \bigcirc \left[ \bigvee_{b=1}^r l_b \right] \quad \text{(a step rule)} \\
\bigwedge_{a=1}^g k_a & \Rightarrow \bigdiamond l \quad \text{(a sometime rule)}
\end{align*}
\]

**Figure 3.2. Rules in SNF**

Generally, we will omit the outer ‘ \( \square \)’ operator and only refer to the different rules, with the understanding that they apply at all moments in time.
SNF rules describe the state of the world in terms of what holds now, at the next moment in time, and at some future moment in time. This is an intuitive view on temporal order, and therefore on dynamic activity. We can now directly execute formulae in SNF [Fis96a]. We create a model as follows: First, the initial rules specify the constraints on the initial moment in time. This allows us to build the initial state. Then, for each state already constructed, we can construct the next one by examining the step and sometime rules, and define the next state such that they are satisfied. That way, we create a model that satisfies the formulae in a forward chaining fashion [FO93, BFG+96].

One more property to note about SNF is that temporal modalities are expressed in terms of $\bigcirc$ or $\Diamond$, with the other operators, such as $\Box, \mathcal{U},$ and $\mathcal{W}$ (as well as corresponding past-time operators) being reduced to the core SNF. Accordingly, we only need to deal with the operators \texttt{start}, $\bigcirc$, and $\Diamond$ in the remainder of this chapter, and indeed this thesis.

### 3.1.3 Example

In order to make \textsc{MetateM}, its implementation and its semantics clearer, we present here a simple example which we will use during the remainder of this chapter. It consists of five simple rules pictured in Figure 3.3. Essentially, it

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{start} $\rightarrow$ $x$</td>
<td></td>
</tr>
<tr>
<td>$x$ $\rightarrow$ $\bigcirc (a \lor b)$</td>
<td></td>
</tr>
<tr>
<td>$b$ $\rightarrow$ $\bigcirc d$</td>
<td></td>
</tr>
<tr>
<td>\texttt{true} $\rightarrow$ $\bigcirc \neg d$</td>
<td></td>
</tr>
<tr>
<td>$a$ $\rightarrow$ $\bigcirc \text{end}$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.3. Simple formulae in SNF form
models a situation where two possible courses of action can be taken (a and b), one of which (b) ends in a contradictory state (in which d and ¬d become true). Specifically, we have one model (Figure 3.4(a)) where a is satisfied in the second state. Another model (Figure 3.4(b)), where b is satisfied at the second moment in time, will be unsatisfiable, because d and ¬d are inconsistent, yet are forced to appear at the same moment in time. We will later see how the implementation deals with inconsistent states.

![Figure 3.4. Possible models for example formulae](image)

### 3.2 MetateM

While MetateM, being based on temporal logic, is simple enough to be directly executed, it clearly lacks the expressivity that other agent logics based on KARO [vL96] or BDI [RG91] offer. Fisher and Ghidini extended MetateM with beliefs [FG99], and later with abilities and confidence [FG02a], bringing it closer to the established BDI formalisations.

Abilities are formalised in a very simple manner. They are fixed over time, and can only range over propositional formulae. While this appears, and indeed is, a rather simplistic way to define abilities, it ensures the resulting logic remains relatively simple.

In order to keep the language lightweight enough to be realistically executable, belief is modelled using a multi-context logic [GG01]. Another motivation was the ability to model a form of resource bounded agents [FG99]. Essentially, belief
contexts behave very much like a standard \textbf{KD45} logic, with the difference that a bound on the depth of belief exploration can be defined. Axiom 4, commonly used to describe positive introspection ($B\varphi \Rightarrow BB\varphi$) states that if you believe something, then you believe that you believe that thing. In order to ascertain the truth value of a given formula, potentially infinite nesting of beliefs have to be explored. Bounded belief allows us to cease exploration of nested beliefs at a given depth.

Confidence, rather than being modelled using a distinct modality, is defined in terms of belief and time. To be “confident in $\varphi$” is expressed as $B\lozenge \varphi$, literally, believing that sometime in the future, $\varphi$ will hold. Defining confidence in this manner serves two purposes. Firstly, the logic remains relatively simple, as no additional modal dimension is added. Secondly, this defined notion allows for flexibility within the logic. For example, an agent can become confident about something because (a) the agent would like it to occur and chooses to be confident, (b) the agent can see how to achieve it (and is able to achieve it), (c) the agent belongs to a group of agents that together can make it happen, or (d) the confidence was communicated to it. Note that being confident in $\varphi$ does not mean that $\varphi$ does have to become true at some point. However, it does allow the agent to reason with this information.

A simple yet fairly powerful example of reasoning using confidence is discussed in [FG02a]. Agents contain the rule

$$A\varphi \land B\lozenge \varphi \Rightarrow \lozenge \varphi \quad (3.1)$$

which states that if an agent has the ability to do $\varphi$ and is confident in $\varphi$, it will eventually make $\varphi$ true. This property is important as it ensures that agents will actively try to make things happen if they are confident in them. Furthermore, this concept can be extended to multiple agents in a very natural fashion, as we will see in Sections 5.2 and 5.5.
In the following definition of the extended temporal logic, we will deviate from
the general thrust of this chapter and immediately introduce a multi-agent version
of the temporal logic with bounded belief (TLBB). The reader may assume the
number of agents to be exactly one to recover a single agent version.

Informally, belief that an agent \( \epsilon \) holds about some set of agents \( \{1, \ldots, n\} \)
is represented in a tree structure, with \( \epsilon \) as root and each node branching into
n subtrees, one branch for each agent’s belief (see Figure 3.5 for a graphical
representation). Each node actually contains a time line that models the world as
believed by the agents. For example, node 21 contains a time line that represents
the world as believed by agent 2 about the beliefs of agent 1 (from point of view
of agent \( \epsilon \).

Before we formally define the logic, we will have to expand a little on multi-
context logics. First, we define a signature \( \Sigma_{TLBB} = (\mathcal{P}, \mathcal{I}) \), where \( \mathcal{P} \) is a set of
propositional constants and \( \mathcal{I} = \{1, 2, \ldots, n\} \) is a set of agents. Now let \( I^k \) be a
set of (possibly empty) strings of the form \( i_1 \ldots i_k \), where all indexes \( i_j \in \mathcal{I} \) and
\( |i_1 \ldots i_k| \leq k, k \geq 0 \). We call any \( \alpha \in I^k \) a belief context. Intuitively, each context
represents a certain nesting of belief operators.

Formally, we define the logic TLBB as an extension of TL. Given the signa-
ture \( \Sigma_{TLBB} \) we define the well formed formulae \( WFF_\alpha \) of TLBB as follows:
any element of $\mathcal{P}$ is in $WFF_\alpha$

- **true**, **false**, and **start** are in $WFF_\alpha$

- If $\varphi, \psi \in WFF_\alpha$, the so are $\neg \varphi, \varphi \land \psi, \varphi \lor \psi, \varphi \Rightarrow \psi$

- If $\varphi, \psi \in WFF_\alpha$, the so are $\bigcirc \varphi, \lozenge \varphi, \Box \varphi, \varphi \cup \psi, \varphi \downarrow \psi$

- $B_i \varphi$, and $i \in \mathcal{I}$ is an atomic formula in $WFF_\alpha$ iff $\varphi \in WWF_{\alpha i}$

- if $\varphi \in WFF_\alpha$ but does not contain temporal or belief operators, then $A_i \varphi \in WFF_\alpha$

The set of well formed formulae $WFF_{TLBB}$ is the union of all sets $WFF_\alpha$. Note here that belief formulae are considered atomic at each level!

The semantics for $TLBB$ are defined as follows. A model for $TLBB$ is defined as $\mathcal{M} = \{ M_\alpha \}_{\alpha \in I}$, where each $M_\alpha$ represents a time line as defined for $TL$. Each $m_\alpha \in M_\alpha$ provides a valuation function $\pi_\alpha$ which assigns truth values to the propositions as well as atomic belief formulae of the form $B_i \varphi$. The temporal part is interpreted in the usual way via the satisfiability relation $|= for the appropriate $m_\alpha$.

In order to arrive at the desired semantical interpretation of beliefs, constraints between the different contexts $\alpha$ and $\alpha i$ are introduced. Informally, if a formula $B_i \varphi$ holds in context $\epsilon$, $\varphi$ has to hold in context $i$ (accessible from $\epsilon$). Therefore, the following constraints are imposed on the model:

if $m_\alpha |= B_i \varphi$ then $m_{\alpha i} |= \varphi$ \hspace{1cm} (3.2a)

if $m_{\alpha i} |= \varphi$ then $m_\alpha |= B_i \varphi$ \hspace{1cm} (3.2b)

if $m_\alpha |= B_i \varphi$ and $B_i \varphi \in WFF_{\alpha i}$ then $m_{\alpha i} |= B_i \varphi$ \hspace{1cm} (3.3a)

if $m_\alpha |= \neg B_i \varphi$ and $B_i \varphi \in WFF_{\alpha i}$ then $m_{\alpha i} |= \neg B_i \varphi$ \hspace{1cm} (3.3b)

Formulae (3.2a) and (3.2b) constrain the multi context logic such that it is equivalent to the modal logic $K$ [BGS98, GG01]. Furthermore, adding the constraints
(3.3a) and (3.3b) forces the logic to satisfy multi context versions of modal axioms 4 and 5 respectively. Because each belief context \( \alpha \) is associated with exactly one temporal model of time, \( m_\alpha \), \( M \) also satisfies multi-context versions of axioms D and fun [FG02a, Ghi99].

Let us now formally define ability. As mentioned earlier, we do not place great demands on abilities. The properties that ability has to satisfy are the following:

- agent \( i \) is able to achieve \( \varphi \land \psi \) if, and only if, it is able to achieve both \( \varphi \) and \( \psi \);
- agent \( i \) is not able to achieve a contradiction; and
- if agent \( i \) can achieve \( \varphi \), or is able to achieve \( \psi \), it is able to achieve \( \varphi \lor \psi \) as well. Note however that this does not hold in the other direction.

We point out that above definition of conjunctive abilities, as used in [FG02a], deviates from the more common definition which uses a simple instead of a double implication used here.

More formally, we define a function \( r : I^k \rightarrow 2^{WFF} \) that associates belief contexts \( \alpha \) with a set of formulae \( \varphi_1, \ldots, \varphi_n \) that agent \( i \) is able to achieve or believed to be able to achieve by other agents. The function \( r \) has to satisfy the following properties:

\[
\varphi \land \psi \in r_i(\alpha) \text{ iff } \varphi \in r_i(\alpha) \text{ and } \psi \in r_i(\alpha) \quad (3.4a)
\]

\[
\text{if } \varphi \in r_i(\alpha) \text{ then } \neg \varphi \notin r_i(\alpha) \quad (3.4b)
\]

\[
\text{if } \varphi \in r_i(\alpha) \text{ or } \psi \in r(\alpha) \text{ then } \varphi \lor \psi \in r_i(\alpha) \quad (3.4c)
\]

\[
\text{if } \varphi \Rightarrow \psi \text{ and } \psi \Rightarrow \varphi, \text{ then } \varphi \in r_i(\alpha) \text{ iff } \psi \in r_i(\alpha) \quad (3.4d)
\]

**Extended Separated Normal Form**

In Section 3.1.2 we saw how arbitrary temporal formulae could be transformed into a set of formulae (in SNF) that could be executed in a forward chaining
fashion. Now, we introduce an extension to SNF that allows arbitrary TLBB formulae to be transformed into SNF$_{BB}$ [FG99].

In TL, a outermost $\square$ operator ensures that propositions associated with (sub-)formulae hold at all moments in time. In order to transfer this technique to TLBB, a new operator $\square^*$ is introduced that allows arbitrary nesting of $B_i$ and $\square$ operators to achieve the effect of a global modality. Essentially, $\square^*$ is defined using fixpoints, following the epistemic version in the temporal time of knowledge as found in [DFW98]. Allowing for that change, SNF$_{BB}$ formulae are of the general form

$$\square^* \bigwedge_i T_i$$

where the rules $T_i$, are in one of the forms in Figure 3.6, and where each $l_b, k_a$ is a literal or ability formula. Note that the only difference to the SNF rules (Figure 3.2) is the belief rule.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbf{start}$</td>
<td>$\Rightarrow \bigvee_{b=1}^{r} l_b$ (an initial rule)</td>
</tr>
<tr>
<td>$\bigwedge_{a=1}^{g} k_a$</td>
<td>$\Rightarrow \bigcirc \left[ \bigvee_{b=1}^{r} l_b \right]$ (a step rule)</td>
</tr>
<tr>
<td>$\bigwedge_{a=1}^{g} k_a$</td>
<td>$\Rightarrow \lozenge l$ (a sometime rule)</td>
</tr>
<tr>
<td>$\bigwedge_{a=1}^{g} k_a$</td>
<td>$\Rightarrow B_i \left[ \bigvee_{b=1}^{r} l_b \right]$ (a belief rule)</td>
</tr>
</tbody>
</table>

Figure 3.6. Rules in SNF$_{BB}$

Multiple Agents

Another important extension of METATEM is concurrent METATEM [FB91, Fis94], where the framework is extended to describe not only single but multiple agents and their interactions. As the second part of this thesis is devoted to
the usage of MetateM within a multi-agent framework, we will here only give a very brief overview, in order to allow the reader to understand some of the concepts that are used in the remainder of this chapter.

In Concurrent MetateM agents execute concurrently and asynchronously, and communicate via broadcasting messages [FB91, Fis96b]. Each agent has an IN and an OUT list of predicates that it listens to and broadcasts, respectively. In the former case, predicates that are in the IN list become true if corresponding messages are received, and in the latter case predicates that are true in a state cause corresponding messages to be broadcast to the other agents. While, initially, the agent space was flat, Fisher and Kakoudakis [FK99] structured the agent space as follows. Each agent, rather than being connected to just one set of other agents, had two sets of agents it was connected with, called Content and Context. The idea was that agents not only were part of a group of agents, but could actually serve as a container for other agents too. The Context set contained agents that were serving as group containers, whereas the Content set contained “members” of the agent.

### 3.2.1 Executing MetateM

In Section 3.1.2, we described how temporal formulae could be executed in general terms. However, the description was non-deterministic, as it assumed that states were created with just the right propositions to find a satisfying model. Here, we will go into more detail, and discuss a deterministic approach.

Given the SNF rule form of present implies future, executing the specification is rather straight forward. Recall from Section 3.1 that temporal formulae are interpreted on a linear model of time comprising of discrete points in time. As the formulae describe either the first state, transitions from the current to the next, or situations that will occur at some future point, the model can be constructed
state by state, in a forward-chaining fashion

METATEM tries to construct a model by executing the set of rules at every moment in time, starting at “the beginning of time”. In this it is different from standard logic programming, as in that case they try to refute the set of rules.

The construction roughly works as follows: At every step in time, METATEM looks at its rules, checks which ones are satisfied in the current state of the model, and collects together the conclusions of those rules. Those, together with outstanding requirements (*sometime* clauses) are used to construct the next state. Eventualities that are still outstanding are passed to the next state, and the circle is closed. If METATEM encounters a contradiction, it backtracks, discarding the current state and trying to construct a new, consistent state using a different (consistent) combination of propositions and eventualities.

**Deliberation**

The order in which different sets of propositions are attempted depends on the following heuristics:

1. make as few literals true as possible
2. make as many eventualities true as possible, starting from the oldest outstanding one.

Note that the requirement of making old eventualities true first guarantees completeness, but there are other orderings possible that also allow for completeness. In [Fis97a], Fisher proposes a function that, during each cycle, re-orders eventualities. While this allows agents to employ more complex deliberation schemes than “oldest one first”, it has some drawbacks. Foremost, the agents’ behaviour is influenced by elements that are not part of the agent’s behavioural description, which is given in SNF rules. Also, the agent is bound by a given ordering function, and cannot change it’s deliberation based on circumstance.
In our system, we propose a built-in function to re-order the eventualities at run-time from within the agent specification. We provide a predicate `prefer/2`, which suggests that the first argument should be attempted before the second one. Using this predicate, together with rules that describe when, and how, a change of order should occur, allows the programmer to provide dynamic deliberation from within the agent description. For example, for a given a list \([a, b, c, d]\) of outstanding eventualities, applying `prefer(a, d)` would not change the list, while `prefer(d, a)` would change the list to become \([d, a, b, c]\). As with Fisher’s approach, this potentially renders the system incomplete, as eventualities might not be tried at any moment in time. However, where previous approaches as in [Fis97a] required the deliberation process to be programmed outside the actual agent program, our approach moves it inside the actual agent description, allowing for a more fine-tuned approach. Sections 6.3 and 7.2 show how `prefer/2` might be used. Note, especially in the latter example, how in multi-agent environments deliberation is handled by re-ordering the eventualities according to information received from other agents.

Note also that the preference network generated by `prefer/2` should provide a linear order on preferences. If it does not, the ordering information is ignored.

Abilities

With respect to abilities, we decided to allow more flexibility by allowing the agent to dynamically ascertain its abilities. Consequently, abilities can be modelled by using a predicate `able/1`, which the agent can make true if it can achieve the argument.
3.3 Implementing METATEM

This section describes our implementation of the METATEM engine. Details of the implementation of multiple agents can be found in Chapter 5. However, we will have to sometimes pre-empt later descriptions and introduce elements of the multi agent implementation, in order to explain some of the choices made here.

3.3.1 Previous Implementations

It should be noted here that there have been some implementations of METATEM in the past, most notably Magenta by Fisher. Magenta is a Prolog based implementation of METATEM that included deliberation functions [Fis97a]. While presenting a working implementation, it had serious drawbacks. Most notably, one had to provide input that an agent received beforehand in form of a Prolog list, together with the rules. Fisher later implemented concurrent METATEM in C++, which allowed for several agents to run in parallel and communicate, but the implementation was never made public. Also, the implementation did not support backtracking of agents.

We felt that a proper implementation of the full METATEM language was necessary in order to further any research into the usability of executable temporal logic and the extensions. The same argument holds for implementing the structured agent space.

3.3.2 The Algorithm

We will now detail how we implemented the actual engine that executes $SNF_{BB}$ formulae. Because the implementation follows closely the structural semantics, we refer the reader to Section 3.4 for a detailed discussion of the algorithms described.
3.3 Implementing MetateM

We execute temporal formulae by creating a temporal model that satisfies the given formulae. This is possible due to the special form of SNF formulae, which all define relations between propositions in the actual world and propositions in the future. Therefore, we can build a model that satisfies the set of rules step by step in a forward chaining fashion. Initially, a (subset of) the literals that are defined in the start rules are made true. Then, for each state, given some literals that hold in that state, we can find all the rules that apply, and construct the next state based on the consequents of applicable rules.

If no consistent next state can be constructed, the current state is discarded and another possible set of propositions is chosen from the last state to construct a next state. The overall algorithm is shown in Figure 3.7, and explained in detail in the following subsections.

repeat
    interpret rule base;
    check consistency of beliefs;
    if consistent
        create DNF, eventualities list;
        choose successor state and add to history;
    otherwise repeat
        remove current state;
    until possible successor state exists;
until either (history empty) or (consistent state and no successor) states exist;

Figure 3.7. General algorithm for executing MetateM

Interpreting Rule Base

In order to create a new state, we first need to identify all rules that are satisfied by the current state. We implement this using the algorithm shown in Figure 3.8, described in detail in Section 3.4.5. Following this algorithm, we end up with a set, $R$, of rules which not only contain the original rules, but also contain rules
for all literals \( p \in P \) that hold in the current state
\[
\text{repeat for all rules } r \in R, \ r = (A \Rightarrow B) \\
\quad \text{if } \sigma(p) \in \sigma(A), \ \text{add } r' = (\sigma(A \setminus p) \Rightarrow \sigma(B)) \ \text{to } R \quad (1) \\
\quad \text{if } \sigma(p) = \neg \sigma(q) \in A \text{ and } q \notin P, \quad (2) \\
\quad \text{add } r' = (\sigma(A \setminus p) \Rightarrow \sigma(B)) \ \text{to } R \\
\text{until no } \sigma(p) \text{ match any antecedent of any rule.}
\]

Figure 3.8. Outline Algorithm for Rule Interpretation

that have smaller antecedents. The rules that have an empty antecedent are said

to fire.

In above algorithm for matching rules, formula (1) matches literals that hold
in the current state. Formula (2) is somewhat more complicated. Essentially, we
need to ensure that rules that have an antecedent that can be satisfied by making
literals true independent of the rules executed in the last state, fire, as long as
satisfying their antecedent is consistent with the current state.

The rules are divided into their different types, namely next, st (sometime),
and bel rules (the start rules are treated as next rules with the difference that
they are only interpreted at the beginning of time). The consequences of next
rules together form a conjunctive normal form (cnf) of literals that have to be
true at the next moment in time; st rules provide a conjunct of literals that need
to be satisfied eventually, and bel rules yield a conjunct of disjunct of literals
that are believed at the current moment in time. We then create a dnf from
the cnf of the consequences of next and st rules, and bel rules respectively,
and add the eventualities to the list of current outstanding eventualities.

We now proceed as follows: First, we check if beliefs are consistent with the
current state. If so, we continue by choosing a successor state; otherwise, we
backtrack.

The next two subsections will describe how beliefs are interpreted, and how,
given a cnf of literals, we can construct the next moment in time.
3.3 Implementing MetateM

Considering Beliefs

While rules generally refer to future states, this is not the case with belief rules, which specify the beliefs of the current state. Therefore, we cannot identify inconsistent states (with respect to beliefs) before we interpret the rule base. Beliefs are interpreted as follows.

First, for each belief modality, a disjunctive normal form is created from the conjunctive normal form extracted from the rule base. Then, for each modality, we create a new time line with, as start state, the literals of current state, plus a disjunct of the belief DNF. We then create the model as we do for the original approach, with the following differences. No interaction takes place (this is only relevant in a multi-agent setting), and no side-effects are executed. Furthermore, we stop creating a model if states repeat. Because there are only finitely many literals, and finitely many rules, we are bound to encounter a recurring state if the program is satisfiable. If we arrive at a recurring state, the belief is consistent, and we return to the next higher model, otherwise the state fails, and backtracking is initiated at the parent level. If no more belief disjuncts are available, we backtrack to the next higher level.

In order to restrain the depth, a counter keeps track of the number of time lines that are created to test consistency of beliefs. If a given number is reached, we assume beliefs to be atomic propositions, following the logic of bounded beliefs [FG99].

Recall that agents can hold beliefs about multiple agents. In each state, each belief modality creates its own time line, to the given depth.

Selecting a Successor State

Given a consistent state, and a set of rules that “fire”, we can now create a successor state. The consequents of NEXT rules that “fire” describe a conjunctive
normal form of literals that has to be satisfied at the next moment in time. We transform this CNF into a DNF — all disjuncts represent a possible combination of literals for the next state. Before we choose such a disjunct, however, we need to do two things. Firstly, we add eventualities to the DNF (discarding inconsistent disjuncts). Secondly, we need to order the disjuncts as follows:

- make as few literals true as possible;
- make as many eventualities true as possible, in the order that eventualities are listed.

The first rule simply stems from the idea that agents try to do as little work as possible. The second is needed in order to ensure that the execution is complete, i.e., it finds a solution if there is one.

**Backtracking**

If a state is not consistent with respect to its beliefs, or if a state has no consistent successor state, we need to backtrack and try a different choice at an earlier point in time. This is generally rather straightforward, as the construction of states leads to an ordered list of possible assignments for successor states. Whenever we choose a successor state, we simple remove the current choice from the list of assignments. If now backtracking occurs, we can simply discard the actual state and again choose the first assignment to create a new successor state. If the list is empty, we need to go one step back and do the same again. In the limit, we arrive either at a history of consistent states, or the history will be empty, in which case no consistent model is found, and we stop execution.

**Completeness**

As the algorithm we employ is very close to the one given in [BFG+96, Chapter 3], we show that our implementation does indeed find a model that satisfies the set
of rules if there is one. The algorithm as described in [BFG+96, Chapter 3] has a cycle as shown in Figure 3.9, while the cycle that is used in our implementation is shown in Figure 3.7 (on page 55). The following paragraphs discuss the parallels between the two algorithms.

```prolog
execute([],[],history):-true.
execute(rules,commitments,history):-
    check-past(rules, history, constraints),
    rewrite(constraints,commitments,exec-form),
    choose(exec-form,rules,cur-constraints,new-comms),
    build-new-state(history,cur-constraints,new-hist),
    execute(rules,new-comms,newhist).
```

Figure 3.9. Prolog algorithm for METATEM

between the two algorithms.

checkpast/3 examines the rules, and returns the consequents of rules that fire given the last state. This corresponds to Interpret rule base of our system.

rewrite/3 rewrites the list of consequents into a DNF, which is what we do in create DNF, eventualities list.

choose/3, build-new-state/3 corresponds to create DNF, choose successor state. Because we order the DNF, choosing, in our case, involves just picking the first disjunct from the DNF. Also, as we are not using prolog-like backtracking mechanisms, our algorithm is somewhat more elaborate to cater for backtracking.

### 3.3.3 Implementation Details

**Abilities**

While we endeavoured to follow METATEM as closely as possible, we chose not to implement abilities as first-class citizens with their own modality. In METATEM,
abilities are modelled as formulae that cannot contain belief or temporal op-
erators, and that have to follow certain axioms (see Formulae (3.4a–d)) that are
mainly concerned with consistency, and otherwise treat abilities just like “normal”
propositional formulae. Furthermore, abilities as defined by Fisher and Ghidini
[FG02a] are constant over time. We can conceivably check before executing a
program whether or not abilities are consistent with respect to each other, and
need not waste resources to constantly monitor consistency of abilities during
execution.

In our framework, abilities are predicates that actually do something. While,
from a logical point of view, literals have no meaning other than truth values
assigned to by some satisfaction relation, programs generally aim at changing
their environment in some way. We therefore allow predicates to be associated
with (outside) actions. Generally we distinguish between internal and external
actions. The former are fixed, and govern the behaviour of agents. The latter are
left to the programmer. Predicates can either just be associated with rules that
fire whenever the predicate is satisfied, but it can also trigger outside code to be
executed, therefore lifting a METATEM agent from an executable specification to
a real world application.

Having side-effects, in the form of external communications, can mean a loss
of completeness. Sending a message to another agent or writing some data on
the screen cannot be undone if the agent encounters an inconsistent state and
needs to backtrack. However, without interacting with the real world our pro-
gramming language is inherently unusable for serious applications. Consequently,
in designing the language, we consciously chose to allow for such incompleteness
while encouraging programmers to work in such a way that communication oc-
curs only when no backtracking will be needed — for example when a solution,
or a planning step, has been found. Thus, typically, agents are programmed to
have a period of internal backtracking (searching), followed by communications with other agents. Thus communication effectively acts as a “cut” operator.

**Programming Agents**

In practice, agents are not only defined by their set of rules, but also by their relation within the multi-agent system. In the actual implementation, an agent framework consists of several elements. Firstly, an *environment file* defines the initial agents, as well as their relation with each other — in other words, the agent’s initial *Content* and *Context* is described. Secondly, each agent is defined by its *agent definition file*, which consists of a header that, for example, defines the agent’s name, the (initial) depth of belief exploration, and abilities, and a set of so-called *rule blocks*. Figure A.2 in Appendix A gives an overview of the full range of settings the header can have.

A rule block is a named set of rules that logically belong together. Agents can load (default) rules from rule files. In order to allow simple individualisation of agents, rules of a rule block always fully overwrite any rules that might have been registered under the same rule block. That way, agents can all load a file with a set of common rules, and re-define behaviours individually in their agent definition file.

Rules consist of predicates that have terms as arguments. Terms can be either functions (that are related to predicates), variables, integers, or strings. We point the reader to Appendix A for in-depth documentation of the implemented system.

**Variables**

METATEM is based on propositional temporal logic. But often we want to work with concepts such as *colour* = \{green, red, blue\}. We can of course create propositions *color_green* . . . in order to express the colour green etc. However, this
also means that we potentially have to repeat rules many times with different colour-propositions.

Instead, we have opted to use *pseudo-predicates* in our language. Rules can contain variables — for example we are allowed to write a rule \texttt{inRoom(X),goTo(Y)->NEXT moveTo(X,Y)}, instead of having to repeat above rule for all possible combinations of rooms. However, this has repercussions as to the execution of the logic. As we deal with a propositional logic, only rules that do not contain any (free) variables are executed. What is more, rules with variables cannot fire “by default”. For example, the rule \texttt{test() \Rightarrow NEXT execute()} can potentially fire at any state that does not specifically satisfy \texttt{not(test())}, potentially making \texttt{execute()} true in the successor state. On the other hand, the rule \texttt{test(X)-> execute()} will only lead to \texttt{execute()} being made true if the predicate \texttt{test/1} was explicitly satisfied (and as only ground predicates can be satisfied in a state, the variable \texttt{X} will be bound to a ground term).

It follows that using variables within rules leads to incompleteness, because the system does not try every possible combination of propositions in order to arrive at a satisfiable model. For example, given a rule \texttt{p(X)-> NEXT end()}, the algorithm would never try to satisfy \texttt{p(X)}, therefore \texttt{end()} would never become true (as a consequence of this rule).

On the other hand, it not only allows for easier programming, but it also opens possible ways of streamlining the execution, as certain rules will only be tried if they “make sense”. In fact, using rules with variables allows to prune the search space considerably. For example, the formula $\varphi \Rightarrow \Box \Box \psi$ translates into the SNF formulae $\varphi \Rightarrow \Box r$ and $r \Rightarrow \Box \psi$. During execution, $r$ can become true at any state, making $\psi$ true in the successor state independently of whether $\varphi$ was true in the preceding state. While this might be correct from a logical point of view, it is often not desirable from a programming point of view.
In order to ensure that the system can indeed provide an accurate execution tree (and so retain completeness), the programmer can either choose not to use variables (which is not desirable), or (s)he can define the domain of the arguments of predicates by adding a rule that always makes a disjunct consisting of all such predicates true. For example, in order to ensure that above rule about the rooms will (potentially) fire at every moment in time, we have to add a rule $\Rightarrow \text{NEXT inRoom(room1); inRoom(room2)}$ (and similarly for the predicate $\text{goTo}$), because then the system will, at every moment in time, make the (grounded) predicate $\text{inRoom/1}$ true, which allows the rule to fire.

From the given example it might be clear that, often, the total exploration of the state-space is not desirable; for the given example, we would not want the agent to change rooms unless specifically asked to do so. Therefore, we do believe that the inclusion (and rather free usage) of predicates and variables adds to the power of the language without adding the computational complexity that a true first order language would entail.

### 3.3.4 Example

We will use the example shown in Figure 3.3 (Section 3.1.3) to create an agent definition file.

We define the name of the agent to be “example”. Furthermore, we do not wish to see debug output of the agent, but we do want to see an execution trace. The predicate name $\text{end}$ is connected to an external ability which is defined in the Java class $\text{Beep}$; Whenever $\text{end()}$ becomes true, the class will be executed (and in this case give an audible sign). Finally, we declare a rule block containing all the rules of the example. In a more complex program, we would use more rule blocks to combine sets of rules that conceptually belong together. We arrive at an agent definition file as given in Figure 3.10.
Name example;
statedebug true;
debug false;
ability end: Beep;

ruleblock:{
  START => x().
  x() => NEXT a(); b().
  b() => NEXT d().
    => NEXT not(d()).
  a() => NEXT end().
}

Figure 3.10. Example of an agent definition file

3.4 Structural Semantics for MetateM

In this section we will present an operational semantics for MetateM processes as described in the preceding section.

We will here only cover single agent semantics. Semantics for multiple communicating agents executing within a structured agent space are defined in Section 5.7. Note, however, that in order to ensure consistency of the whole semantics, some of the structures presented now will contain elements that are not relevant for the single agent case.

Operational Semantics

Semantics are a way to assign meaning to a (syntactic) entity. While there are many different types and styles of semantics, some of which we described earlier, this section will centre around operational semantics for executing MetateM agents.

In an operational semantics we are concerned with how to execute programs, rather than just the results of the execution. Generally, we can distinguish between structural operational semantics, which describes individual steps of the
3.4 Structural Semantics for MetateM

computation, and natural semantics, that describes how overall results of executions are derived [NN99].

The following few sections will describe a structural operational semantics based on Plotkin’s transition style [Plo04], though we do at times revert to natural descriptions of processes, if we feel that it helps the reader to better understand the overall system.

We omit proofs of termination, completeness, and soundness. We note that termination is relatively easy to prove as Equations (3.15a-c) are termination conditions which by construction will always be reached. Soundness and completeness are more complex, and are left as future work.

**Abstractions**

We will develop the semantic interpretation of the agent system based on an abstract syntax of the actual programming language. As mentioned in Section 3.3.3, an agent system consists of an environment file which defines the structure of the system, agent definition files that define the behaviour of the agent, and possibly rule files that provide behaviours that agents can include in their description.

Our abstract syntax uses an agent description, which contains the rule base of the agent, as well as structural information, namely the Content and Context of the agent. Furthermore we store the depth to which beliefs can be expanded in the agent description. One of the results of using this abstract syntax is that certain structural elements used in the actual programming language disappear. Most notably, we will not encounter the variable $Self$, as it is replaced by the name of the agent. We will also not deal directly with rule blocks, as we assume that the rules of an agent do not change over time.
Notation

We will use the following notational conventions in the remainder of this section. We will use **roman bold font** to denote basic sets used in the language for the semantics, **typewriter type** to describe elements of the object language, **Capital Italic Fonts** to denote set, list, or tuple variables within the semantics, and **italic fonts** to denote variables of the semantics. Tuples are written with ⟨angles⟩, sets with {curly brackets}, and lists in [square brackets]. The empty set is denoted by ∅, the empty list by [], and [Head|Tail] refers to the first element (Head) and the remainder of the list (Tail). We use + to denote concatenation of lists, and A[x/y] to denote that x is replaced by y within A.

We combine elements using tuples, and we will often abbreviate tuples with a variable name and refer to elements of the tuple by using subscripts. For example, to refer to the set Rules of the tuple A = ⟨AgentName, Rules⟩, we write Rules_A.

### 3.4.1 Basic Definitions

In this section we define the basic entities used to provide the agent descriptions. These definitions will hold in the single as well as the multiple agent case.

**Definition 1 (Constants)** We have a small set of constants, consisting only of the constants Content and Context.

**Definition 2 (AgentNames)** AgentNames are objects describing the name of an agent. We write AgentNameSet to denote a set of elements of AgentNames.

**Definition 3 (Terms)** We define terms in the usual way. It includes strings, natural numbers, agent names, functions (that we relate to predicates), sets of terms, and variables.
Definition 4 (Predicates) We define predicates in the usual way. The union of predicates and negated predicates we call **Literals**. Generally, terms do not include variables. If predicates or literals are allowed to have variables as arguments, we denote this by subscribing the set of predicates/literals with the word \textit{var}. We define \textbf{LiteralList} to be a list of elements of \textbf{Literals}.

Definition 5 (Rules) Let \textbf{Rules} be a tuple \(\langle \text{LHS}_{\text{var}}, \text{RHS}_{\text{var}}, \text{type} \rangle\), with \textit{type} \(\in\) \{\text{start, next, some, bel}_i\} as start, next, sometime, and belief rules respectively (\(i\) defines the agent with whom the beliefs are associated), and \(\text{LHS}_{\text{var}}\) (resp. \(\text{RHS}_{\text{var}}\)) being a set of literals. If the type of the rule is irrelevant to the task at hand, we omit the last element of the tuple and write \(\langle \text{LHS}_{\text{var}}, \text{RHS}_{\text{var}} \rangle\). We define \textbf{RuleSet} to be a set of \textbf{Rules}.

Definition 6 (Beliefs) Let \textbf{Beliefs} be a set of named lists of sets of literals. Each List \(B_i \in \text{Beliefs}\) of sets of literals is a representation of the disjunctive normal form of beliefs that are held about agent \(i\). We use \textbf{BeliefDNF}_i to denote the list of sets of literals that is associated with agent \(i\).

Definition 7 (Agent Description) We define \textbf{Agents} as the set consisting of tuples \(\text{Agent} = \langle \text{AgentName}, \text{Content}, \text{Context}, \text{Rules}, \text{BeliefDepth} \rangle\), where \text{AgentName} is an element of \textbf{AgentNames}, while \text{Content} and \text{Context} are \textbf{AgentNameSets}. \text{Rules} is of type \textbf{RuleSet}, and \text{BeliefDepth} is a natural number. For readability, we will often omit elements in the tuple that are irrelevant to the task at hand, and for example describe an agent description as a tuple \(\langle \text{AgentName}, \text{Rules} \rangle\).

Single Agent Definitions

In addition to above definitions, we also need some definitions that will only hold in the single agent case.
Definition 8 (States)  Let States be a set of tuples $\langle P, Ev, B, DNF \rangle$, where $P$ is a set of elements of Literals and $Ev$ (of type LiteralList) are lists representing predicates and eventualities respectively; $B$ is of types Beliefs and consists of a set of named lists of sets of literals representing the possible beliefs that the agent holds about other agent’s beliefs. $DNF$ is a list of sets of literals representing a DNF representation of possible assignments of literals that can be true at the next moment in time. $DNF$ stores all possible combinations of predicates that have to be true at the next moment in time.

Note that in order to describe a state of our model, the elements $P, Ev, B$ would be enough. The other elements of the state are needed to describe possible successor states, and therefore to enable us to provide a deterministic semantics.

Definition 9 (Model)  Our model we define to be a (potentially infinite) list of states $S$. We will write History to refer to the list.

We now have sufficient building blocks to start and assign meaning to a MetaTEM agent program. We will define the semantics using infix transition functions ‘$\rightarrow$’ (following [Pl04]) of increasing detail. Starting by defining a general transition $\rightarrow_h$ from an agent description to a history of states, we continue to describe the history function in terms of applying state transition functions $\rightarrow_s$ which in turn depend on interpreting rules $\rightarrow_r$ and choosing successor states $\rightarrow_c$. Interpreting rules depends on single rules $\rightarrow_{r^i}$ and those on the interaction of rules and predicates, which we describe by use of a function $\rightarrow_{sub}$, again breaking it up in two further functions $\rightarrow_{mkSt}$ and $\rightarrow_{rem}$. Note that the semantics we establish is deterministic, and will define a path through an ordered tree that represents the possible executions of the agent definition.

Definition 10 (Semantic Function)  In order to assign meaning to our agent, or to connect it to a model, we define a semantic function $S : \text{Agents} \rightarrow \text{Model}$ as follows:
3.4 Structural Semantics for MetateM

\[ S(Agent) = \begin{cases} \text{History} & \text{if } Agent \rightarrow_h \text{History} \\ \text{FAIL} & \text{otherwise} \end{cases} \quad (3.5) \]

The above function (Definition 3.5) tells us that given an agent definition, we either get a (potentially infinite) list of states describing the execution of the agent, or the agent definition is illegal. Note that the correctness of the agent definition does not make any statement about the program itself — it is still very possible that the program is unsatisfiable, in which case we end up with an empty history.

3.4.2 History Level

The most general way to look at the execution of an agent program is in terms of the history. Let us have a closer look at the function \( \rightarrow_h \). Starting from the empty history, we create new states based on the preceding one, until a fix-point is reached. Assuming for the moment a state transition function \( \rightarrow_s \), we can define the history function \( \rightarrow_h \) as follows:

**Definition 11 (History Function)** \( \rightarrow_h : \text{Agents} \rightarrow \text{Model} \) is an infix function from an agent description to a list of states, defined as

\[
\langle Agent, [] \rangle \rightarrow_s^* \langle Agent, History \rangle \\
\text{Agent} \rightarrow_h \text{History}
\]

Note that the resulting history can very well be empty, if the execution fails, or describe an infinite but recurring computation as \( * \) denotes the transitive closure of individual \( \rightarrow_s \).

3.4.3 State Level

The above definition is rather straightforward. It becomes more interesting though if we go down a level and examine the detail of the function \( \rightarrow_s \). It
has, as input, an agent description and a history, and should output an agent
description and a new history. For simplicity we do not allow changes in the
agent description here. In order to be able to define $\rightarrow_s$ however, we need to
explain in greater detail how META TEM agents construct their model.

Creating States

Recall that we aim at constructing a deterministic semantics. Therefore, instead
of being able to just choose the “right” successor state, we have to constructively
describe how to choose such a successor state. We accomplish this by computing
all possible successors for each given state, and expanding the resulting tree struc-
ture following an heuristic of choosing states that satisfy as many eventualities
as possible while keeping the number of satisfied literals as small as possible.

As mentioned earlier (see Def. 8), our framework adds additional data that
describes possible future states in order to be able to provide a deterministic
semantics.

First of all, we keep a list of all possible sets of satisfied literals for the successor
state that can follow from the rule base in the form of a formula in disjunctive
normal form (DNF); secondly we keep the set of eventualities that follows from
the rule base. The latter is needed as different sets of literals (within the DNF)
might satisfy different the eventualities. We order the disjuncts within the DNF
relative to the list, $Ev$, of outstanding eventualities as follows:

**Definition 12 (Ordering)** We define a total ordering $\sqsubseteq_{Ev}$ on sets of literals
$DNF_i$ relative to a list of literals $Ev$ as follows:

1. $DNF_1 \sqsubseteq_{Ev} DNF_2$ iff $Ev = \emptyset$, and either $|DNF_1| < |DNF_2|$ or $|DNF_1| =
   |DNF_2|$ and $hash(DNF_1) < hash(DNF_2)$. In words, if $Ev = \emptyset$, then the
   DNF that forces smaller set of literals to be true is preferred (in case of
equal size, an arbitrary but consistent order given by the function $hash$ is
2. Given $Ev = [l|\text{Tail}]$, let $DNF_1 \sqsubseteq_{Ev} DNF_2$ iff $l \in DNF_1$ and $l \not\in DNF_2$.
Thus, if exactly one of the sets of literals contains the first literal of the list of eventualities $Ev$, it is preferred.

3. Given $Ev = [l|\text{Tail}]$, $DNF_1 \sqsubseteq_{Ev} DNF_2$ iff $l \in DNF_1$, $l \in DNF_2$, and $DNF_1 \sqsubseteq_{\text{Tail}} DNF_2$. Here, if both sets contain the first element of $Ev$, the the sets are again compared with respect to the remainder of the list of eventualities.

4. Given $Ev = [l|\text{Tail}]$, and $l \not\in DNF_1$, $l \not\in DNF_2$, then $DNF_1 \sqsubseteq_{Ev} DNF_2$ iff $DNF_1 \sqsubseteq_{\text{Tail}} DNF_2$. Finally, if neither contain the first element of $Ev$, they are compared again with respect to the remainder of the list.

Given the ordered list of literals that can be satisfied in the next state, we are able to create a successor state simply by choosing the first set of conjuncts from the DNF representation as the successor state’s set of satisfied literals; the new list of eventualities is obtained by removing any satisfied literals from the current list. The consistency of such a new state depends on the set of chosen literals and beliefs.

The last sentence needs some explanation. Generally, METATEM creates new states in a forward chaining fashion. New states are created by executing NEXT rules and choosing a (consistent) set of literals that will define the next state. Similarly SOME rules are interpreted and influence the choice of literals. BEL rules however do not influence future states but rather the current state. This means that we can choose a consistent set of literals for a new state, but we will only know whether the literals are consistent with respect to the current belief set after having interpreted the rule base and analysed the beliefs that follow. Note that checking consistency with respect to belief literals is not trivial, because
we need to expand belief contexts in order to determine the valuation of belief formulae. We refer the reader to Section 3.4.6 where we will describe in detail how this is accomplished.

Assuming, for the moment, that we have means to determine the consistency of created states, we must provide a mechanism for back-tracking if a state turns out to be inconsistent. Because we have to expand the state (i.e., interpret the rules) to create possible future states as well as to determine consistency, we specify two distinct operations to define a state transition \( \rightarrow_s \) — the interpretation of rules at the one hand \( \rightarrow_r \), and the modification of the history on the other hand \( \rightarrow_c \). Essentially, \( \rightarrow_r \) will expand the current state to contain information about all possible successor states, and \( \rightarrow_c \) will choose a successor state.

**State Rules**

We now give the precise interpretation of the function \( \rightarrow_s \) in terms of the functions \( \rightarrow_r \), which describes how rules are interpreted, and \( \rightarrow_c \), which chooses a successor state.

**Definition 13 (State Function)** Let the function \( \rightarrow_s : \text{Agents} \times \text{Model} \rightarrow \text{Agents} \times \text{Model} \) be a function from an agent description and a history to an agent description and history following the transition rules:

\[
\begin{align*}
\langle \text{Agent}, S \rangle \rightarrow_r^{\uparrow} S', \quad & [S'|\text{History}] \rightarrow_c \text{History'} \\
\langle \text{Agent}, [S|\text{History}] \rangle \rightarrow_s \langle \text{Agent}, \text{History'} \rangle & \quad \text{if } S' \text{ consistent} \quad (3.7a) \\
\langle \text{Agent}, S \rangle \rightarrow_r^{\uparrow} S', \quad & \text{History} \rightarrow_c \text{History'} \\
\langle \text{Agent}, [S|\text{History}] \rangle \rightarrow_s \langle \text{Agent}, \text{History'} \rangle & \quad \text{if } S' \text{ inconsistent} \quad (3.7b) \\
\text{Agent} \rightarrow_{st} S, \quad & \langle \text{Agent}, [S] \rangle \rightarrow_s \langle \text{Agent}, \text{History'} \rangle \\
\langle \text{Agent}, [\text{]} \rangle \rightarrow_s \langle \text{Agent}, \text{History'} \rangle & \quad \text{at the initial state} \quad (3.7c)
\end{align*}
\]
Definition 13 ensures that we only expand the history if the current state is consistent with respect to the belief set. In case of an inconsistency, the current state is discarded, and the preceding state is taken as a choice point. We refer the reader to Section 3.4.6 for a more precise definition of consistency. Equation (3.7c) initialises the history by means of the start transition function $\rightarrow_{st}$, that interprets the start rules and creates an initial $DNF$ representation. As start rules are interpreted in a similar way to next rules, but are only used at the beginning of time, we omit them here for readability and leave it to the reader to expand the semantics given to make the necessary transition function explicit.

**Choosing a Successor State**

Let us deal with the choice function $\rightarrow_c$ before defining how rules are interpreted. We said earlier that $\rightarrow_c$ chooses a successor state, but we should rather say that the choice function modifies the history. The difference between these two statements is that while modification can often consist of adding another state to the history, this is not necessarily the case, as we might have chosen a “wrong” state earlier and subsequently ended up in a state with no successor state, in which case $\rightarrow_c$ would have to backtrack. More formally, we can distinguish three different ways of modifying the history.

- The current state has at least one possible successor state, i.e. it has at least one set of literals in its DNF, so we choose the first set of literals from the DNF, remove it from the list (and thereby modify the state), and create a new state (Equation (3.8a)).

- The state’s DNF is empty, in which case we have no possible successor state; we backtrack by removing that state and choosing a different extension of the remainder of the history (Equation (3.8b)).

- The state’s DNF is empty, and there is no earlier state. In that case we
stop with an empty history (Equation (3.8c)).

Formally, we define the choice function as follows:

**Definition 14 (Choice Function)** Let \( \rightarrow_c : \text{Model} \rightarrow \text{Model} \) be a function from history to history, following the following rules:

\[
S = \langle P, Ev, B, [P'|\text{DNF}^+] \rangle, \\
S' = \langle P', \text{merge}(Ev, P'), [], [], [] \rangle, \\
S'' = \langle P, Ev, B, \text{DNF}^+ \rangle \\
\text{if } \text{DNF}_S \neq [] (3.8a)
\]

\[
[S|\text{History}] \rightarrow_c [S', S''|\text{History}] \\
\text{if } \text{DNF}_S = [] (3.8b)
\]

\[
[S|\text{History}] \rightarrow_c [] \\
\text{if } \text{DNF}_S = [] (3.8c)
\]

Here, the \( \text{merge} \) removes duplicates from the list of eventualities, as well as eventualities that are satisfied by the successor state.

In formula (3.8a), \( S \) represents the current state, \( S' \) the successor state, and \( S'' \) the current state with the extension chosen for successor state removed.

Note that (3.8a) is the only transition that actually adds a new state to the history. Also, it removes the first element of the \( \text{DNF} \) (the set of literals chosen to be satisfied at the next state) from the current state (\( S \), which becomes \( S'' \)). This is necessary to ensure our semantics is constructive and deterministic. Effectively, the \( \text{DNF} \) represents all possible successor states. \( \rightarrow_c \) always takes the “preferred” next state, which is the first element of the \( \text{DNF} \). By removing chosen elements, we automatically traverse the tree of possible successors from left to right.
3.4 Structural Semantics for MetateM

3.4.4 Rules Level

While the above definitions describe the behaviour of a MetateM agent on the level of states, we will now detail how we construct possible successor worlds from the original agent description. Note that, at this stage, we are no longer interested in successor states themselves, only in the “expansion” of states in order to determine possible future states (as well as consistency with respect to belief states).

Agent descriptions are interpreted by examining their rules and extracting beliefs, eventualities, and possible valuations for a successor state.

Intuitively, a rule fires if the literals within a state satisfy the antecedent of the rule. Depending on the type of rule, the consequent is then added to a set of possible new literals (in case of a next rule), beliefs (bel rule), or a list of eventualities (st rule). Note that in case of a next rules, we end up with a set of sets of literals that have to be satisfied at the next moment in time. This set of sets is a conjunctive normal form (CNF) representation, as the consequent of each next rule is a disjunction, and rules are combined by conjuncts. Accordingly, we will convert this CNF into a DNF, and order it as in Definition 12. Bel rules also create CNF’s, for each agent i. Therefore, we will also have to transform the CNF’s for each belief context to belief DNF’s for each agent. The ordering we apply is similar to the next case, while assuming the set of eventualities being empty. Thus, we get an ordering of the disjuncts according to their cardinality.

Merging Rule Applications

Single rule applications only provide parts of a state. Therefore, we will have to combine the result of applying all rules in order to complete the state. To this end, we define a function which merges partial states that are created by the interpretation of single rules. As this function is very similar to performing a
simple union on sets, we will write it in inline form.

**Definition 15 (Merging Partial States)** Let $\cup : \text{States} \times \text{States} \rightarrow \text{States}$ be a function that merges two state descriptions as follows:

\[
(P, Ev_1, B_1, DNF_1) \cup (P, Ev_2, B_2, DNF_2) = (P, Ev, B, DNF)
\]

where $Ev = Ev_1 + (Ev_2 \setminus e)$, for all $e \in Ev_1$, and $B = \{B_i | B_i = \text{merge}(B^i_1, B^i_2, \emptyset) \}$ for all $B^i \in \{B_1, B_2\}$ (noting that $B^i$ is assumed to be the empty set if no BEL rule for agent $i$ fired, i.e. no $B^i$ was defined). Finally, $DNF = \text{merge}(DNF_1, DNF_2, Ev)$. The function $\text{merge}$ merges two DNF’s and imposes the order from Def. 12 while discarding inconsistent conjuncts.\(^1\) Note that NEXT rules, as well as belief rules, create disjunctive normal forms of literals. We leave the precise definition of $\text{merge}$ to the reader.

Given the function to merge partial states into one, we can now define how the interpretation of a set of rules in a state is achieved by interpreting each rule of that set, and merging the results of single rule applications into a fully expanded state.

Recall that we abbreviate complex entities such as agent descriptions and states while still referring to their elements by subscribing them appropriately (for example, $P_S$ refers to the set $P$ of state $S$). We can now provide the intermediate step of interpreting rules by defining (abusing our notation slightly)

\[
\forall r \in \text{Rules}_A : \langle r, P_S, Ev_S \rangle \rightarrow_r S' \quad S = \bigcup_r S'
\]

(3.9)

Here, in order to compute the next state $S$, all rules are interpreted by $\rightarrow_r$, each yielding a partial state description. These partial states are then combined to create the new next state. For example, if we had two firing rules $p \Rightarrow q$ and $r \Rightarrow s \lor t$, the two partial DNF’s would be $\{q\}$ and $\{\{r\}, \{s\}, \{r, s\}\}$, which would then be combined to yield $\{\{q, r\}, \{q, s\}, \{q, r, s\}\}$.

\(^1\)We use the third argument to pass a list of literals on to the ordering function, which takes the role of $Ev$ in the definition.
### 3.4 Structural Semantics for MetateM

#### Single Rules

Rules are interpreted by matching literals that hold in the current state to the left hand side of a given rule. If we can match all literals of the left hand side (where negative literals are assumed to be satisfied unless they contain free variables or clash with positive literals), the rule is said to fire, and the right hand side is added to the appropriate set resp. list (in case of eventualities). The following description will not depend on more detailed transition functions, though we will give an alternative semantics later that will go into more details.

Given a substitution $\sigma$, a rule $r = \langle LHS_{var}, RHS_{var}, type \rangle$ (where $LHS, RHS$ can contain variables), a set $CNF$, a state $S = \langle P, Ev, B, DNF \rangle$ and a function $to\text{DNF}$ that transforms a set of sets representing a conjunctive normal form into a corresponding set of sets representing a disjunctive normal form:

\[
\forall \sigma \text{ s.t. } P \models \sigma(LHS_r) : \\
\text{add } \sigma(RHS_r) \text{ to } CNF.
\]

Let $DNF_S = to\text{DNF}_{Ev}(CNF)$

\[
\langle r, P, Ev \rangle \rightarrow_r S
\]

if $type_r = \text{NEXT}$ \hspace{1cm} (3.10a)

\[
\forall \sigma \text{ s.t. } P \models \sigma(LHS_r) : \\
\text{add } \sigma(RHS_r) \text{ to } Ev_S.
\]

\[
\langle r, P, Ev \rangle \rightarrow_r S
\]

if $type_r = \text{ST}$ \hspace{1cm} (3.10b)

\[
\forall \sigma \text{ s.t. } P \models \sigma(LHS_r) : \\
\text{add } \sigma(RHS_r) \text{ to } CNF.
\]

Let $B_i^r = to\text{DNF}_B(CNF)$

\[
\langle r, P, Ev \rangle \rightarrow_r S
\]

if $type_r = \text{BEL}_i$ \hspace{1cm} (3.10c)

We again abuse the notation slightly to represent the fact that belief rules about an agent $i$ that fire add their right hand side (which represents a disjunct of literals) to the appropriate Belief$DNF$ associated with agent $i$. As the function $to\text{DNF}$ takes as argument (in its subscript) a list of eventualities, we use the
empty set in Equation (3.10c) to ensure ordering according to cardinality only.

3.4.5 Sub-Rule Level

Instead of using a high level logical description of how to interpret rules by means of the function \( \rightarrow_r \), we can go into more detail. At this level, we now follow the implementation rather closely.

Intuitively, we interpret single rules by creating a list of partially matched rules (from the single rule) by matching literals against the left hand side, and, in case of a match, apply necessary substitutions and add the modified rule, now with the substitution applied and the matching element of the left hand side removed. After applying all literals in this way, we end up with a list of rules derived from the application of the literals to one rule. We now collect all rules that have an empty left hand side (as these represent rules that “fire”), and combine their right hand sides to arrive at the result of interpreting a single rule. More precisely:

**Definition 16 (Single Rule Interpretation)** Given a rule \( r \), a list of rules \( Rules \) and a set of literals \( P \), we interpret \( \rightarrow_r \) using two helper functions \( \rightarrow_{sub} : RuleList \times LiteralList \rightarrow RuleList \) and \( \rightarrow_{mkSt} : RuleList \times Literals \times Literals \rightarrow States \), which interpret single rules and combine the result of interpreting the rules respectively:

\[
\begin{align*}
\langle [r], P \rangle & \rightarrow_{sub} Rules, \quad \langle Rules, P, Ev \rangle \rightarrow_{mkSt} S \\
\langle r, P, Ev \rangle & \rightarrow_r S 
\end{align*}
\]  

(3.11)

where \( \rightarrow_{sub} \) is defined as:

\[
\begin{align*}
\langle Rules, p \rangle & \rightarrow_{sub}^1 Rules^*, \langle Rules^*, P \rangle \rightarrow_{sub} Rules' \\
\langle Rules, [p|P] \rangle & \rightarrow_{sub} Rules'
\end{align*}
\]  

(3.12a)

\[
\langle Rules, [] \rangle \rightarrow_{sub} Rules
\]  

(3.12b)
Equations (3.12) recursively apply all literals of \( P \) to a list of rules (initially only containing the original rule). The application of single literals by means of the function \( \rightarrow^{1}_{\text{sub}} \) is described in the following definition.

**Definition 17 (Applying Literals to Rules)** Given a list of rules, as well as a single literal, we “apply” the literals to the rule as follows:

\[
\begin{align*}
\text{Rules}^* &= \text{Rules} + \langle \sigma(LHS_r)\backslash q, \sigma(RHS_r), \text{type}_r \rangle, \\
\langle \text{Rules}^*, p \rangle \rightarrow^{1}_{\text{sub}} \text{Rules}' &
\quad \text{if } \exists q \in \text{LHS}_r \\
&\quad \text{s.t. } \sigma q = p \\
\langle [r|\text{Rules}], p \rangle \rightarrow^{1}_{\text{sub}} [r|\text{Rules}'] &
\quad \text{otherwise}
\end{align*}
\]

These equations are a recursive description of the process by which rules are “expanded” against literals. Equation (3.13a) adds a new rule to the list of rules if the literal can be matched against one of the literals of the left hand side of the rule, (3.13b) ignores rules if no such match can be found, and (3.13c) is the “stop condition”.

We are left with the definition of the function \( \rightarrow_{\text{mkSt}} \). While this appears to be straightforward, its precise description is rather involved. Intuitively, the function \( \rightarrow_{\text{mkSt}} \) has to perform two steps. Firstly, we remove all rules that do not have an empty left hand side. However, as mentioned earlier, we can only initially assume that literals not explicitly true are false. In order to include rules whose left hand side is not specifically true or false we need to ensure that literals on the left hand side of rules that are not inconsistent (that is, there is no matching literal in \( P \)) are removed and the remaining rules are kept. This step we specify using
the transition function $\rightarrow_{\text{rem}}$, which will return a set of rules whose left hand sides are empty. Note that though we added some syntactic sugar by allowing variables, we only can make rules true that do not contain free variables.

Secondly, the type of the rule determines where the right hand side ends up. Without giving precise definitions, we assume the following transition relations:

- $\rightarrow_{\text{nx}}$ from rules to sets of sets of literals, this extracts the right hand side of the given rules of type NEXT — representing a conjunctive normal form — and returns a set of set of literals representing a (unordered) disjunctive normal form.

- $\rightarrow_{\text{st}}$ from rules to a list of literals, this concatenates the right hand side of the given rules of type ST and returns the list (as eventualities are kept in a list, rather than a set).

- $\rightarrow_{\text{b}}$ rules to sets of sets of literals, this extracts the right hand side of the given rules of type $\text{BEL}_i$ — representing a conjunctive normal form — and returns a set of named lists of sets of literals representing disjunctive normal forms for the beliefs of each agent $i$.

We now arrive at the following definition:

**Definition 18 (Extracting States from Rules)** Given a list of rules and a list of predicates, we create a (partial) state description as follows:
And finally

\[
\langle \text{Rules}, P \rangle \rightarrow_{\text{rem}} \text{Rules}', \\
\forall r \in \text{Rules}' \text{ s.t. } \text{type}_r = \text{BEL} : \text{Rules}_B \rightarrow_b \text{B}_S, \\
P_S = P, \text{Ev}_S = \text{Ev}
\]

\[
\langle \text{Rules}, P, \text{Ev} \rangle \rightarrow_{\text{mkSt}} S
\]  

(3.15c)
Above rules are to be read as follows. For each type of rule (start, sometime, next), the appropriate rule is chosen.

3.4.6 Interpreting Beliefs

In the preceding section, we described how an agent builds its history by choosing successor states, specifically by imposing an order on the possible successor states, and subsequently choosing consistent ones. While we have mentioned that consistency depends not only on satisfied literals but also on the agent’s beliefs, we have not yet explained how the beliefs of an agent are interpreted. In this section, we re-define the state function (Def. 13) to formally describe consistency with respect to beliefs in a state.

Belief Contexts

In METATEM, beliefs are interpreted using belief contexts. This means that in order to ascertain the truth value of a believed literal, a new model, or context, is created in which the belief modality is stripped from the belief literals, and the model is expanded to test for satisfiability. However, in METATEM, the belief contexts are defined as to closely resemble KD45, for example we assume that if $B\varphi$ then $BB\varphi$. Accordingly, we would have to create new many models in order to establish whether a belief holds. As METATEM is designed to be executable, this is often not feasible. Fisher and Ghidini therefore introduced the concept of bounded beliefs [FG99].

In our implementation, we evaluate beliefs as follows. We create (for each agent $i$) a new model (or context) whose start state consists of the literals and eventualities of the original state, as well as a disjunct of literals added to the set of true literals chosen from the DNF of the associated agent. The model is created (executed) until it either fails, loops, or satisfies the set of rules. With
each level of model we decrease the belief depth counter. When it reaches zero, we stop and assume that the model can be satisfied. More formally, we adapt Definition 13 by introducing a function \( \rightarrow_b: \text{Agents} \times \text{Model} \rightarrow \text{Boolean} \) as follows:

**Definition 19 (State Function with Belief)** Let the function \( \rightarrow_s: \text{Agents} \times \text{Model} \rightarrow \text{Agents} \times \text{Model} \) be a function from agent description and a history to an agent description and history following the transition rules:

\[
\begin{align*}
\langle \text{Agent}, S \rangle & \rightarrow^+_r S', \\
\langle \text{Agent}, S' \rangle & \rightarrow_b \text{result}, \\
[S'|\text{History}] & \rightarrow_c \text{History'} & \text{if result} = \text{true} & \text{and } S \text{ consistent} \\
\langle \text{Agent}, [S]\text{[History]} \rangle & \rightarrow_s \langle \text{Agent}, \text{History}' \rangle & \text{otherwise} & (3.16a) \\
\text{History} & \rightarrow_c \text{History'} & \text{otherwise} & (3.16b)
\end{align*}
\]

Function \( \rightarrow_b \) creates new models for each agent about which beliefs are held, and expands them following the transition rules that we have given so far. The only difference is that, instead of seeding the history with literals derived from start rules, the initial state is created by adding appropriate belief literals (chosen from the DNF associated with the agent whose belief state is expanded) to the literals of the ancestor state. If the construction yields non-empty models, the original state is considered to be consistent with respect to the set of beliefs. Note that the newly created models themselves expand belief contexts up the the depth specified in the agent description.

**Definition 20 (Belief Function)** Abusing notation slightly, we define the belief function \( \rightarrow_b: \text{Agents} \times \text{Model} \rightarrow \text{Boolean} \) as follows:
Agent′ = Agent[beliefdepth/beliefdepth − 1],
∀Bi ∈ BS, S∗ = S[P_S/(P_S ∪ bi)], where bi ∈ Bi
⟨Agent′, [S∗]⟩ →∗ s History,
result = true if History > 0 for all Bi,
false otherwise

⟨Agent, S′⟩ →b result if beliefdepth > 0 (3.17a)
⟨Agent, S′⟩ →b true otherwise (3.17b)

For brevity, use natural semantics in above definition, in that we assumed that
the algorithm picks a bi which does yield a non-empty model. For operational
semantics, we would need to introduce two further transition functions, one that
cycles through the beliefs associated with different agents, and another to cycle
through the possible assignments of beliefs for the given agent.

3.4.7 Example

We will now take the example as given in Figures 3.3 and 3.10 and show how the
program behaves in the operational semantics presented here. Because of space
and clarity considerations, we will only present a subset of the application of the
transitions, which will show the workings of the semantics.

Recall that the example program consisted of the following rules:

START => x().
x() => NEXT a(); b().
b() => NEXT d().
    => NEXT not(d()).
a() => NEXT end().
3.4 Structural Semantics for MetateM

Using above rules, we arrive at an abstract agent description $A$ that is extracted from the program:

$$\langle \text{example}, \emptyset, \emptyset, \emptyset, \{\langle \emptyset, x() \rangle, \text{start}\}, \langle x(), \{a(), b()\}, \text{NEXT}\rangle,$$
$$\langle b(), d(), \text{NEXT}\rangle, \langle \emptyset, \neg d(), \text{NEXT}\rangle, \langle a(), \text{end}(), \text{NEXT}\rangle \rangle, 3 \rangle \quad (3.18)$$

We can now start to interpret the program. At the beginning of time, only START rules are interpreted.

Given the rule $r = \langle x, \{a, b\}, \text{NEXT} \rangle$ which yields a state $\langle r, x, \emptyset \rangle$, we arrive at a (partial) state $S = \langle x, \emptyset, \emptyset, \{\{a\}, \{b\}, \{a, b\}\} \rangle$ as follows. As Equation (3.20) shows, the transition rule $\rightarrow_r$ (see Equation (3.11)) can be interpreted by applying all literals that hold in the state to the rule set $R$. In our example, we have only one literal, $x$, that needs to be applied, so $\rightarrow_r^1$ is called once, with $\langle \langle x, \{a, b\}, \text{NEXT} \rangle, x \rangle$ which yields the set of rules $\{\langle x, \{a, b\}, \text{NEXT}\rangle, \langle \emptyset, \{a, b\}, \text{NEXT}\rangle\}$. Next, we check the resulting set of rules whether there are rules that fire by default, i.e. there are rules whose LHS is consistent with the literals that are true in the current state. In this case, no rules change, and, using the transition $\rightarrow_{\text{mkSt}}$ (see Equation (3.21)), we arrive at a (partial) State $S = \langle x, \emptyset, \emptyset, \{\{a\}, \{b\}, \{a, b\}\} \rangle$. Together with the rules that fire by default (via $\rightarrow_{\text{rem}}$), the state is extended to contain the following DNF: $\{\{b, \neg d\}, \{a, \neg d\}, \{a, b, \neg d\}, \ldots, \{a, b, d, \neg d\}, \{a, b, d, \text{end}, \neg d\}\}$. Assuming that the function $\text{merge}$ prefers $\{b, \neg d\}$ to $\{a, \neg d\}$ (according to Definition 12, we assume some order on predicates of the same cardinality, so $b$ could be preferred to $a$), the history gets extended to a next state, which turns out to have no consistent successor states. So we backtrack and choose $\{a, \neg d\}$ in state 1, which then leads to a consistent state.
\[ \forall r \in R_S : \langle r, P_S, Ev_S \rangle \rightarrow S^\prime, S^\prime = \cup_r S^\prime \]

\[ S = \langle P, Ev, B, \{P'|DNF^+\} \rangle \]

\[ S'' = \langle P', merge(Ev, Ev'), [\text{],} \theta, [\text{]} \rangle \]

\[ S^{m} = \langle P, Ev, B, DNF^+ \rangle \]

\[ \begin{align*}
  \langle A, S \rangle & \rightarrow^*_S \langle S', S'' \rangle \\
  \text{[}S'[H]\text{]} & \rightarrow^*_c [S'', S'''\mid H] \\
  \langle A, [] \rangle & \rightarrow^*_S \langle A, H \rangle \\
  A & \rightarrow^*_h H 
\end{align*} \]

Equation (3.19) shows how the transitions are traversed in order to create the history. At the first moment in time, only one rule fires. In (3.19), we assume the application of the high level definition for rule interpretation (3.10a)–(3.10c) (where, because start rules are interpreted as NEXT rules, we use Equation (3.10a)), and arrive at initial state \( S_0 = \langle \emptyset, \emptyset, \emptyset, \{\{x()\}\} \rangle \). We now use the choice transition relation \( \rightarrow^*_c \) (Equations (3.8)) to create the next state in time \( S_1 = \langle \{x()\}, \emptyset, \emptyset, \{\{x()\}\} \rangle \), and arrive, after execution of the START rule, at the history \( H = [\langle \{x()\}, \emptyset, \emptyset, \{\{x()\}\} \rangle, \langle \emptyset, \emptyset, \emptyset, \{\{x()\}\} \rangle] \).

Now, we go through the next iteration of the state transition \( \rightarrow^*_s \). Note that as we are not at the beginning of the execution any more, we will have to run through all rules (bar the START rule that we will omit from this point on), all of which fire as well, because for all rules \( r \) holds that \( x \models \sigma(LHS_r) \).

We will use the rule \( \langle x, \{a, b\}, next \rangle \) to demonstrate the behaviour of the sub-rule transitions. Equation (3.20) shows the (partial) proof tree for the sub-rule part of the transition system.
\[ R' = (\emptyset, \{a, b\}, \text{next}) \]

\[ (R', x) \xrightarrow{\text{sub}} [\emptyset, \{a, b\}, \text{next}] \]

\[ \langle x, \emptyset \rangle \rightarrow_{\text{mkSt}} \langle x, \emptyset \rangle \]

\[ (R', x) \xrightarrow{\text{sub}} [(\emptyset, \{a, b\}, \text{next}), (x, \{a, b\}, \text{next})] \]

\[ \langle x, \emptyset \rangle \rightarrow_{\text{rem}} [(\emptyset, \{a, b\}, \text{next}), (x, \{a, b\}, \text{next})] \]

\[ \forall r \in R': \text{compute } S \text{ based on } r \]

\[ \langle x, \emptyset \rangle \rightarrow_{\text{mkSt}} \langle x, \emptyset \rangle \]

\[ \langle x, \emptyset \rangle \rightarrow_{\text{rem}} [(\emptyset, \{a, b\}, \text{next}), (x, \{a, b\}, \text{next})] \]
Part II

Multiple Agents
Chapter 4

Background — Multi Agent Systems

In the previous chapters, we have seen a variety of different ways to specify agents. However, agents arguably only start to become interesting once there is more than one. In this chapter, we will investigate the issues that arise when talking about multiple agents. We start out with a broad overview over the elements that play a role in multi-agent systems, such as concurrency (Section 4.1) and communication (Section 4.2). We then focus on multi-agent systems themselves, and discuss multi-agent formalisms (Section 4.3) and behaviours (Section 4.4), and conclude with an overview over some multi-agent frameworks in Section 4.5.

4.1 Concurrency

In order to produce a multi-agent system, we need to have more than one agent. While this in itself sounds (and is) trivial, the question of how one runs several agents in parallel is not.

Generally, one can either have a (dedicated) machine per agent, or one can
run several agents in parallel on one machine. While the former is straight forward (not taking into account issues such as communication, which we consider later), the latter needs computers and operating systems capable of doing so. In modern operating systems, there are two ways of doing things in parallel. We can distinguish between processes and threads. While both approaches allow programs to execute more than one instruction (apparently) at the same time, they are different in the way those executions are organised. Processes are what we might call first-class citizens from an OS perspective. Processes “live” in their own address space, and have their dedicated (virtual) registers. An operating system imitates parallelism by switching through the running processes, instantiating the processors registers with the data associated with the process, and executing (some) instructions before saving the state and switching to the next process [Sta03].

Threads, on the other hand, live within a process. Rather than having their own space, all threads within a process share the same address space. They are “light-weight” as it takes much less time to switch between threads than between processes. Furthermore, the fact that they share an address space allows for very efficient communication methods such as shared memory.

If parallelism is simulated on a single machine, we can distinguish between fine- and coarse grained parallelism. The former interleaves (single or very few) instructions, generally divided by some physical measurement such as the number of instructions, while coarse grained parallelism is generally defined by logical measures such as the execution of a procedure, independent of the time it takes to actually execute it [CSG98].

Agent frameworks that are based on, for example, the Java programming language generally use the built-in methods to achieve parallelism by using threads, which corresponds to fine-grained parallelism. Other languages, such as ConGolog
4.2 Communication

In order for multiple agents to work together in any form, they need to communicate. Depending on the framework, they can use methods that differ in various ways, such as the type of information sent, reliability of the communication, addressing, and many more. We will here highlight some of those issues.

Agents running within the same address space can employ very fast and reliable communication using shared variables [Dij65], or blackboard communication, where agents can all access a common memory area. While, in the former case, agents themselves have to ensure data integrity of the shared variable, for example by means of semaphores [Dij68] or monitors [Hoa74], blackboards by design only allow one agent to modify the shared structure at a time. It might be clear to the reader that this method assumes that agents do have access to some common address space, which is not the case if agents are distributed over a wide area network.

Often, communication between two entities is described by means of channels. The $\pi$-calculus introduced channels as named entities which link processes [MPW92]. While reducing computation to the communication of names over links, as the $\pi$ calculus does, is very abstract, the concept of channels as a named link between two processes can be found throughout the literature. However, Busetta et al. define channels as a stream of messages that may be listened to by many agents simultaneously, similar to a radio channel [BDN02].
More common, essentially because they are much more flexible, are communication methods that allow agents to interact via networks, independent of shared address spaces or similar. This is generally accomplished using message passing. Messages usually have a sender and one, or more, recipients. Depending on the protocol which is used to transmit the message, transmissions can be synchronous, i.e. the sender receives immediate confirmation of the message being received, or a-synchronous, where the sender continues with its computation while the message travels through the network. In the latter case, delivery is generally not guaranteed.

Most agent systems employ communication methods that are based on the protocol standards established by the internet, such as TCP (Transmission Control Protocol) [Pos81b] or UDP (User Datagram Protocol) [Pos80] over IP (Internet Protocol) [Pos81a]. TCP allows for reliable communication, but is more costly, while UDP guarantees neither delivery nor the order of messages being retained. Messages can be sent to either single recipients (using point-to-point (p2p) addressing), a group of recipients (using multi-cast addressing (IGMP)), or broadcast over a segment of the network (generally broadcasts are only sent to all members of a subnet) [Sta03].

All addressing methods have their advantages and disadvantages. For example, p2p communication over TCP/IP provides synchronous reliable and (potentially) secure communication. However, if many agents interact as a group, the amount of messages being sent grows exponentially, potentially swamping the network with duplicate messages. In addition, agents need to know and remember the addresses of all agents that should receive particular messages. Broadcasting on the other hand allows messages to be sent to all agents (assuming they all live within the same subnet) very simply. However, we cannot exclude agents from
receiving messages. Furthermore, if many agents engage in pairwise communication (using broadcasting), we again saturate the network — assuming a switched network, p2p communication is sent exclusively on the wires that form the actual path from sender to receiver, while broadcast sends each message over all wires. With multi-cast, agents need to register as members of certain groups in order to receive messages being sent to the multi-cast group. This requires much more book-keeping, as well as a understanding of which agent group is represented by which multi-cast group. Furthermore, multi-cast messages are often not allowed outside subnets.

4.3 Multi-Agent Formalisms

On a more abstract level, sophisticated communication between agents is often formalised and represented using speech acts [Aus62, Sea69, CL85, CL90]. Speech Act theory is based around the idea of treating utterances as actions on a par with physical actions, in that they are performed with the intention of bringing about a change in the environment, usually a change of the listener’s mental state.

Austin [Aus62] proposes that we call the utterance itself — the words artifici-
ally divorced from their social context — a locution and then explore the locution in context, as a complex relational speech act; as an illocution, what we intend to do in saying something; and as a perlocution, the effect on our listener that we want to have by saying something. Thus, for example, the adult who says to a child, “I’d love to see your drawing,” might be describing a state of mind (locution), promising to look at the drawing (illocutionary force), and attempting to make the child feel good, building the child’s self-esteem (perlocutionary effect).

Treating utterances as actions entails that speech acts may fail just as physical actions may fail. For example, an utterance might not bring about the desired change in the listener’s mental state.
While, initially, Speech Acts were examined from a linguistic point of view, they soon attracted computer scientists working in artificial intelligence (for example [CL85]). Cohen and Levesque defined speech acts using a formal logic with four modal operators, denoting belief (BEL), Mutual belief (MBM), goals (GOAL), and execution of acts (AFTER) (interpreted as in a dynamic logic). They then defined a multitude of properties, for example knowledge ($((\text{KNOW} \ x \ p) \equiv p \land (\text{BEL} \ x \ p))$), or sincerity, which they defined as $(\text{SINCERE} \ x \ p) \equiv (\text{GOAL} \ x \ (\text{BEL} \ y \ p)) \Rightarrow (\text{GOAL} \ x \ (\text{KNOW} \ y \ p))$, meaning that an agent is sincere with respect to $p$ if whenever his goal is to get someone else to believe $p$, his goal is in fact to get that person to know $p$ [CL85].

In order for agents to be able to communicate seamlessly, so-called agent communication languages (ACL’s) have been developed, which were meant to allow agents from different platforms to interact with each other on a high level [Sin98, LFP99]. The most prominent agent communication languages are KQLM [FMFM94, FF94], FIPA-ACL [FIP97, PM99], and ACPL [vE00]. Essentially, ACL’s provide the agent with a set of (speech) acts that convey information about the agent’s own mental state with the objective of changing the mental state of the communication partner.

Communication acts are comprised of several layers. Most central is the actual content of the message, being expressed using some agreed-upon language, for example first-order logic. Secondly, a particular attitude towards the informational content of the message is expressed in the form of a speech act. Examples of speech acts are tell to express that the content of the message is believed to hold, untell to express that the content should not be believed to hold, or ask to ask whether the content is believed to hold by the communication partner. Finally, an outer layer deals with the mechanics of the communication, including information about sender, receiver, channel, and more.
Once multiple agents can communicate, the question arises as to what they should talk about. As we saw in Chapter 2, (rational) agents are typically defined in terms of their mental states, i.e. their beliefs and intentions. Within a multi-agent system, agents not only can reason about their own beliefs and intentions, but also about those of other members of the system. Instead of just reasoning about their own beliefs ($B_{\text{self}}\varphi$), agents can also reason about beliefs that other agents might hold ($B_{i}\varphi$) and even about beliefs they believe other agents to hold about other agents ($B_{i}B_{j}\varphi$).

The next natural step is to enable multiple agents to communicate (parts of) their internal state. That way, not only can agents act based on their own internal state, but they can establish joint beliefs, intentions, and goals.

Levesque et al. [LCN90, CL91] presented a joint intentions framework, which extends the traditional BDI approach by characterising a team’s mental state. A collection of agents jointly intends a team action if the team members are jointly committed to completing the team action, while mutually believing they were doing so. A joint commitment is defined as a joint persistent goal (JPG). To enter into a joint commitment, all team members must establish appropriate mutual beliefs and commitments. This is done through an exchange of ‘request’ and ‘confirm’ speech acts.

More specifically, a joint persistent goal between two agents $X$ and $Y$ is achieved as follows [CLS97]. $X$ performs a request to $Y$ to do $A$, which leads to both agents mutually believing that $X$ has a persistent weak achievement goal (PWAG) that $Y$ will do $A$. According to Cohen et al. this is already enough to establish a joint persistent goal, as it establishes all the pre-conditions of $A$ being a JPG, which are (a) that the agents must mutually believe that $A$ has not been done, and (b) that both parties want it done, both requirements are established by the PWAG. However, generally an assenting answer by $Y$ to do $A$ is required as well.
in order to establish the JPG. Once the joint goal has been either achieved, is impossible, or is irrelevant, either agent asserts that any of those situations has occurred, and the joint persistent goal is discharged.

The commitment protocol synchronises the team, in that all members simultaneously enter into a joint commitment towards a team task. In addition, all team members must consent, via confirmation, to the establishment of a joint commitment goal — thus a joint commitment goal is not established if any team member refuses. In this case, the group of agents might

- form a team without the dissenting agent,
- modify the joint goal to accommodate the agent, or
- negotiate with the dissenting agent to try to change its behaviour.

Extending the joint intentions framework with teams, Tidhar [Tid99] designed a multi-agent logic which not only includes JPG etc, but provides for many more operators, such as \textit{SUBTEAM}, \textit{COMMAND}, \textit{COMMS}, and more. In this system, one can define axioms that govern the behaviour of agents within teams, and between teams themselves. For example, \textit{COMMAND} denotes that a team is “commandeering” the other; axioms that define the behaviour of two two teams $\omega$ and $\omega'$ in such a relationship are for example $\text{MBEL}(\omega', \phi) \land \text{COMMAND}(\omega, \omega', \phi) \Rightarrow \text{MBEL}(\omega, \phi)$, which states that if team $\omega'$ has a commandeering relationship with $\omega'$, and the team holds a mutual belief $\phi$, then so will the commandeered team $\omega$ [Tid99].

Another approach is used by the SharedPlan model of Grosz and Kraus [GK96], which is based around the mental attitude of “intending that”. The concept is defined via a set of axioms that guide a teammate to take action, or enter into communication that enables or facilitates its teammates to perform assigned tasks.

Both Cohen \textit{et al}. and Groszet \textit{et al}. approach the issue of multi-agent systems
from the perspective of the individual agents, i.e. the behaviour of a team, or multi-agent system is explained in terms of their component agents. Another view is, for example, taken by Searle [Sea90] who argues that the behaviour of a system cannot be explained in terms of its components. So-called *holonic* agents fall under that header. Holonic agents [FSS03] are based on theories in holonic manufacturing [Dee94, HMS94]. Holonic agents can form *holons*, which to the system appear as normal agents. (We note that this is similar to the approach we adopt later in Chapter 5.)

In more recent years, the need to address organisational issues of multi-agent systems has become apparent, and more research is being done in that area (see for example [ZJW04, Fer99, Sch03, NPC+03]). Here, agents are mapped onto an abstract structure that often represents (human) organisations, such as organisations within a company, but also organisational structures that are used to represent task-relations. The actual form of an agent organisation can be anything from strictly hierarchical structures to graph-like structures. For an overview of different organisational structures we refer to [Sch03] and [HL04].

An important concept of organisations is that of a *role*. In an organisational structure, agents are not so much defined by their identity or abilities, but rather by their abstract role. Agents can take multiple roles, and roles can be taken by multiple agents, allowing for robustness of the system. For example, when designing a museum, we can devise different roles, such as a tour-guide role, and a cashier-role. Both roles, however, could be assumed by your PDA, which then fulfils the latter role by deducting some money from your mobile account, and the former by downloading and displaying information relevant to different exhibits that you are standing close to. On the other hand, the role of sales-agent concert tickets for *Pink Floyd* could (at least initially) very well be played by many agents in order to cope with the many connections being established by fans.
4.4 Multi-Agent Behaviours

Even with multi-agent systems that are based on agents capable of communicating and establishing joint beliefs and intentions, it is by no means obvious how agents can achieve the goals that researchers had (and still have) in mind for them. In order to avoid agent technology suffering the same fate that Artificial Intelligence did two decades ago, when researchers promised results but were unable to deliver, different types of agents must be able, independently and in open environments, to communicate and work together.

We have seen theories about team-building where teams are defined in terms of shared intentions or plans [CL91, GK96]. What team-building is based on is some kind of negotiation between members in order to establish the team or the joint goals and intentions. In most general terms, the concept of negotiations between agents is relatively simple. Given a set of agents, a set of possible outcomes, and a preference relation over the outcomes for each agent, negotiation comes down to finding the outcome that is most preferred by all negotiation partners. However, not only is there no such outcome guaranteed to exist, it is by no means clear how the agents would find it as the number of possible outcomes is too large to be simply enumerated [WP00]. Furthermore, agents would generally not want other agents to know the details of their preference relation.

Simple, but pervasive, examples of negotiation protocols are the Contract Net [Smi80] or the Match Maker algorithm [SDW97]. A simple contract net protocol consists of a manager that issues a call for proposals, and a set of bidders that react to the proposal. The manager, after receiving all bids (or reaching a time-out) chooses from the bids made. However, simple protocols inherently assume reliable communications and the availability of all communication partners during a protocol run. The former can be ensured by using appropriate channels, such as TCP/IP or CORBA, but the latter has to be dealt with on the protocol level.
An auction is a method of allocating scarce goods, a method that is based upon competition. A seller wishes to obtain as much money as possible, and a buyer wants to pay as little as necessary. An auction offers the advantage of simplicity in determining market-based prices. It is efficient, in the sense that an auction usually ensures that resources accrue to those who value them most highly, and ensures also that sellers receive the collective assessment of the value. While the general idea is simple, there exist many different auction types, all of which have different advantages and disadvantages. For an overview over different auction protocols see, for example, [Rey96].

One aspect that simple protocols do not take into account is the concept of trust or reputation (for an overview on trust in multi-agent systems see [RHJ04]). Trust can be modelled from two sides, the individual agent or the underlying system. In the former case, agents can reason about strategies, motivations, and information gathered about other agents in order to influence negotiation techniques or negotiation partners, while the latter case forces agents to abide by set rules by imposing conditions that would “punish” agents if they lie, using reputation to further future interactions, or by imposing standards as to the goods that are negotiated about.

Simple negotiation protocols and auctions not only cover services as their objects, but can also allow the agents agree to cooperate or build structures in order to accomplish certain objectives. We can distinguish between different types of such structures, namely coalitions and teams. Coalitions in general are goal-directed and short-lived; they are formed with a purpose in mind and dissolve when that need no longer exists, the coalition ceases to suit its designed purpose, or critical mass is lost as agents depart. They may form in populations of both cooperative and self-interested agents [KG02].

Teams, on the other hand, consist of cooperative agents that have agreed to
work together on some specified goal (see, for example, [LCN90, Tam97]). While
members of coalitions act for their own interest, teams aim at achieving what is
best for the group.

As agents are used to model organisational structures, they often use the
concept of roles, where agents are not only defined by their abilities, but by the
role they hold within a structural environment. For example, agents working
together as a team might elect one agent to act as an intermediate between the
team and the outside world.

4.5 Multi-Agent Frameworks

The abstract BDI architecture has been implemented in a number of systems,
some of which we have discussed in Section 2.2. The first generation is typified
by the Procedural Reasoning System (PRS) [GI89], developed by SRI Internatio-
nal in the mid-1980s. dMARS [Kin93, dKLW98], built in the mid-1990s by the
Australian Artificial Intelligence Institute in Melbourne, Australia, is a successor
system. dMARS has been used as a development platform for a number of techno-
logy demonstrator applications, including simulations of tactical decision-making
in air operations [Tid99] and air traffic management [RG95]. AgentSpeak(L)
(which we discussed in Section 2.2) [Rao96], being the successor for dMARS, has
been the focus of more recent research.

Not mentioned in Section 2.2 was JACK, a Java based agent framework that
also implements the BDI approach [BRHL99]. JACK has gone through several
iterations, with the current one offering extensions that deal with role-based teams
[HRHL01]. As JACK is commercial, its performance in terms of communication
and execution is superior to many academic implementations. However, the se-
mantical basis of the BDI part of the implementation is not really clear.
While JACK realises teams on top of their agent structure, other agent programming languages are explicitly based on some notion of teams, such as Ferber’s MadKit, or Tambe’s TEAMCORE.

MadKit [GF00] is a multi-agent system programmed in Java that is based on a notion of roles, groups and agents (the basis of which is the AALAADIN framework [FG98]). In their framework, an agent is merely a communicating entity which plays roles within groups. Groups define the types of roles that are allowed, and can refuse agents to assume certain roles. The AALAADIN model is an organisational tool that allows agents to exist in different roles and groups, all of which can use different languages and expressions. For example, a computer scientist can at the same time be a tennis player, cook, and musician [FG98]. All those roles require different languages and concepts. AALAADIN defines groups as consisting of atomic agents, so while agents can be member of many groups, groups may not be member of other groups. As agents are not further defined, the groups and their definition of the interaction of roles is the basis of systems developed in AALAADIN.

TEAMCORE [PTCC99] is based on work carried out on STEAM [Tam97], which itself is based on the joint intention theory [LCN90], as well as borrowing some aspects of ShardPlans [GK96]. A STEAM agent’s knowledge about teams is based on three elements: coherence preservation, which requires agents to communicate in order to ensure coherent initialisation and termination of team plans, e.g. a team member must inform others if it uncovers crucial information that would render a team plan un-achievable; monitor and repair, which ensures that team members will substitute for other critical team members who have failed in their roles; and selectivity of communication, which is about weighing communication costs and benefits in order to avoid excessive communication in the team. The SharedPlans part in STEAM can be found in the construction of
hierarchical structures that are similar to SharedPlan theory, especially partial SharedPlans.

Joint intentions of teams are organised using a team operator, a piece of software that ensures communications between team members and that synchronises joint persistent goals. In order to remove this single point of failure, the monitor and repair system allows a team to re-configure itself, i.e. re-distribute roles and designate another team operator in order to fulfill its goal.

TEAMCORE itself provides a wrapper around agents which provides “team-capabilities”, allowing agents to be deployed in teams without having to be able to modify their code or domain knowledge. This addresses an important issue in the practicability of agent theories and multi-agent systems — the ability to use legacy code and/or interface with traditional software and databases.

The framework that has arguably focused most on “agentifying” legacy code is IMPACT [ESR99, ES99, ESR00], which basically is a wrapper language with which interactions between different pieces of legacy code can be described, and combined. One important entity in IMPACT is a so-called “code-call”. A code-call is a tuple consisting of a variable and a predicate describing a function call on some specified domain. Together with software states which describe what objects the embedded software is managing, and a set of action atoms, which describe the actions that the agent can execute, they provide a basis of the language.

Actions can be embedded in a set of deontic operators such as permitting and forbidding, obligations and waiving of obligations, and an operator stating that the agent brings about the action. These are called “action status” atoms. Programs consist of a set of rules, where the antecedent is a conjunction of either code-call atoms or action status atoms, and the consequent is an action status atom. Rather than providing a fixed semantics for their language, they provide a
set of possible semantics to interpret the programs. IMPACT has also been extended to deal with time [DKS01] and uncertainty [DKS02]. While TEAMCORE, for example, allows agents to interface with legacy code, it does not provide a strict semantics for dealing with interactions between the agent layer and legacy code, while IMPACT does exactly that.

For example the code-call \texttt{oracle:select(emp.rel,salary,>\text{150000})} denotes a domain (agent) oracle, and executes the select query. Using such code-calls, we can construct conditional code-calls using logical connectives and variables, such as \texttt{in(X, oracle:select(emp.rel,salary,>\text{150000})) \land in (Y, face:findpicture(mugshotdb,X.name))}. Here, the output of the select-query is stored in \texttt{X} and used to find all pictures of the found people in a different database [ESR99]. This example already shows that code-calls are generally impossible to evaluate if variables are not instantiated.
Chapter 5

Groups of METATEM Agents

While in Chapter 3 we detailed single agents based on METATEM we will, in this chapter, describe the multi-agent-framework, based on METATEM agents, that we have developed.

We will start out by outlining the structure of our framework, following this with an in-depth discussion of the merits of such an approach. After investigating different ways of making use of this structure, we will provide the reader with a more theoretical foundation, after detailing implementation issues and choices, giving a structural semantics of our multi-agent system.

As we have seen in Chapter 3, METATEM agents are defined by a set of temporal and modal formulae, and execute this definition by essentially creating a model in a forward-chaining fashion. In the multiple agent scenario, agents execute their specification asynchronously. They communicate via message passing, which we will describe in Section 5.1.1. Connections to object-oriented approaches are discussed in Section 5.2. Rather than living “side-by-side” agents exist within the framework in a graph-like structure (Section 5.1). Not only can agents move through the structure (Section 5.4), the structure itself can be used to store various data, as detailed in Section 5.3. Also, parts of the structure can have different policies (Section 5.5), allowing for different grouping behaviours. We
finish the chapter by providing details of the implementation (Section 5.6), and a structural semantics for the multi-agent case (Section 5.7).

## 5.1 Structured Agent Space

As we have seen, there exist numerous agent frameworks, some based on logical formalisms, other more geared towards industrial applications. However, most of them assume an environment in which agents either broadcast to the whole set of agents, or employ point-to-point communications. Our framework on the other hand is based on a structured agent environment.

The mechanism we use to structure the agent space is “a group”. However, rather than introducing a new entity, we identify the notion of agents and groups. This means that agents not only can be member of groups, they also can contain other agents, serving as a group for member agents. As agents are essentially programs, they can have intricate rules governing the behaviour of member agents. Agents can be members of several group agents, and may contain an arbitrary number of agents. Membership is dynamic, i. e. agents can join and leave groups, and even create new groups which can be permanent or transient.

Note that, in the remainder of this thesis, we will use the term “agent” and “group” interchangeably, depending on what aspect of an agent we want to focus on. It is vital for the understanding to realise that this is for ease of writing only — we are always talking about a MetateM agent.

In short:

- agents can be members of several groups;
- groups can contain several agents;
- groups can be nested arbitrarily;
- groups are dynamic — agent can join and leave dynamically;
- groups can have different membership policies;
• agents can contain other agents; and finally,
• agent $\equiv$ group!

We create this structure by giving each agent two sets through which it can communicate, called $Content$ and $Context$. Agents that are members of an agent are situated within the agent’s $Content$, while groups the agent is member of are contained within the agent’s $Context$. Figure 5.1 shows three different ways of representing the resulting structure. Agent $A$ is representing the environment, which contains three agents $B, G, I$, which in turn contain other agents. Note in particular agent $C$ which is member of two groups, namely $B$ and $G$.

Atomic agents are agents with an empty $Content$, and atomic groups are agents with no behavioural rules (other than the ones necessary to allow agents

<table>
<thead>
<tr>
<th>Agent</th>
<th>$Content$</th>
<th>$Context$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>${B, G, I}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>B</td>
<td>${D, C}$</td>
<td>${A}$</td>
</tr>
<tr>
<td>C</td>
<td>$\emptyset$</td>
<td>${B, G}$</td>
</tr>
<tr>
<td>D</td>
<td>${E, F}$</td>
<td>${B}$</td>
</tr>
<tr>
<td>E</td>
<td>$\emptyset$</td>
<td>${D}$</td>
</tr>
<tr>
<td>F</td>
<td>$\emptyset$</td>
<td>${D}$</td>
</tr>
<tr>
<td>G</td>
<td>${C, H}$</td>
<td>${A}$</td>
</tr>
<tr>
<td>H</td>
<td>$\emptyset$</td>
<td>${G}$</td>
</tr>
<tr>
<td>I</td>
<td>$\emptyset$</td>
<td>${A}$</td>
</tr>
</tbody>
</table>

Figure 5.1. Different views of the system structure
to enter and leave the groups).

We note that depending on the application, different properties of the groups structure might have to be enforced, such as acyclicity, symmetry, or transitivity. The general structure we present here is non-transitive, non-symmetric, and acyclic. However, we do not enforce those properties. This, and a detailed analysis of the influence of different properties on the structure is left to future work.

5.1.1 Communication

Structuring the agent space is only interesting if agents interact, i.e. communicate via message passing. Historically, agent frameworks employ either broadcasting or point-to-point message passing to communicate. More recently, multi-casting is employed as well, for example by Busseta [BDN02, BMRL03].

We employ a multi-cast like communication. Receivers are designated by their names. Moreover, receivers have to be in either Content or Context.¹

In our framework, agents use a simple language to define a set of receivers. While the set of (immediate) receivers has to be contained within the union of Content and Context, agents can specify elements out of this set by naming (sets of) agents, using set descriptions Content or Context, but also set operations such as union, intersection, and exclusion.

More specifically, agents send messages to a set of agents using the predicate send(Set,Message), where Message is the message to be sent, and Set is either a set S of agent names, one of the constants in \{Content, Context\}, the special variable $Self$, or a set-expression $S_1$ union $S_2$, $S_1$ intersection $S_2$, or $S_1$ without $S_2$, where $S_1, S_2$ are set-expressions.

The variable $Self$ allows an agent to reference its own name. While theoretically, we could program agents using their names explicitly, using a variable

¹We have extended the language to use a third set, which we describe in Section 5.6.2
enables the sharing of code between agents.

Furthermore, send predicates can be nested, allowing agents to route messages through the agent space. We will highlight two examples here.

**Group Send**

Agents are grouped for a multitude of purposes, some of which we will highlight in the next section, but it is fair to assume that agents within a group will need to communicate with each other. We call this *group send*, and can implement this as pictured in Figure 5.2. Agent $b$, in order to communicate with its peers, sends a message to $a$ with the request to send a message to $a$’s *Content*. The command to do so is `send(a, send(content without b, Message))`. Note that agent $b$ itself will not receive the message it sent.

On a side note, it should be mentioned that because we are talking about agents and not objects, it is possible for agent $a$ to refuse to forward the message. If we constructed agents that were living in a potentially hostile environment,
agent $b$ could include itself in the message, in order to learn whether or not $a$ indeed forwarded the message. Of course, there is no guarantee that agent $a$ did not forward the message to everybody besides $b$!

**Recursive Send**

Another typical send action would be to broadcast a message to all agents within the agent space. In our framework, rather than hard-coding such a send action, we instead define a rule that forwards a message recursively through the agent space using the following equivalence:

$$
\text{sendAll}(\text{Set}, \text{Message}) \equiv \text{send}(\text{Set}, \text{Message}) \land \\
\text{send}(\text{Set}, \text{send}(\text{Set} \setminus \text{Self}, \text{Message}))
$$

By excluding the sending agent $\text{Self}$ from the set of receivers, we ensure that no message is sent through the agent space continuously. However, above definition does allow for agents to receive a message more than once, if the graph is cyclic. In order to send a message to all agents, $\text{Set}$ is set to “Content union Context”. Of course, agents can also just address all “lower” or “higher” agents, by using just Content or Context, respectively.

### 5.2 Object Oriented Programming

One can view agent oriented programming as an extension of object oriented programming [Sho90]. Whether or not one subscribes to this view, it should be clear that a successful agent programming language should subsume (most of) the features that object oriented programming provides. This section will show how we can simulate object oriented features within our system.

Object Oriented programming extends procedural programming with (at least) the following features:
5.2 Object Oriented Programming

- Inheritance
- Encapsulation
- Polymorphism
- Abstract classes

5.2.1 Inheritance

Inheritance in object oriented programming entails that objects inherit methods and their accompanying data structures from parent objects.

The agent system as such provides a natural way to inherit abilities (methods). Each agent has its “super agent” (a.k.a. super class) in its context, and the following rules within its rule base (for readability purposes we omit some details and write A resp. A? to abbreviate predicates able and isAble respectively):

\[ \neg A(\varphi) \land B \diamond \varphi \Rightarrow \text{send}(\text{Context}, A?(\varphi)) \]  
(5.1a)

\[ \text{receive}(A?(\varphi)) \land \neg A(\varphi) \Rightarrow \text{send}(\text{Context}, A?(\varphi)) \]  
(5.1b)

\[ \text{receive}(A?(\varphi)) \land A(\varphi) \Rightarrow \text{do}(\varphi) \land \text{send}(\text{origin}, \text{answer}(\phi, \psi)) \]  
(5.1c)

\[ \text{receive}(A?(\varphi)) \land \neg A(\varphi) \land \]  
(\text{Context} = \emptyset) \Rightarrow \text{send}(\text{origin}, \text{answer}(\phi, \text{error}))  
(5.1d)

Above formulae define a basic inheritance of abilities, which correspond to, for example, Java methods. Note however that variables are not “inherited”. This can be adapted, at least for basic types, by requiring agents to add certain attribute value pairs to their knowledge base when they are added to a superclass’s content. For example, there could be a message makeVar(“xValue”,2). Furthermore, the above rules assume (for simplicity) that all “abilities” of the agent are indeed public.

Formula (5.1a) tells the agent to ask its context if it encounters an ability it does not have itself (but that it requires). Formulae (5.1b) to (5.1d) define the
reaction of agents that might get a request for an ability $\varphi$. If they are not able to
do $\varphi$, they re-send the message “upwards”, following (5.1b). In the case they do
have the requested ability, they execute it and send the result to the requesting
agent (5.1c). Finally, if there are no ancestors, the agent returns with an error
message (5.1d).

### 5.2.2 Encapsulation

As agents are capsules of data and rules by design, we get this defining notion of
object oriented programming for free.

### 5.2.3 Polymorphism

Polymorphism entails that objects can override methods they might have in-
erited from their ancestors. For example, given a 2d point object with a method
dist(point) that computes the distance between two points, and an extension
that deals with 3d points, we can re-program the dist function to take the new
dimension into account. The programmer does not need to know whether she
deals with a 2d or 3d point, as polymorphism ensures that the methods can be
extended to deal with new situations.

In fact, example formulae (5.1a) to (5.1d) already provide for polymorphism,
as the agent will use its own ability if it has it, or the one of the “closest” agent
if it does not.

### 5.2.4 Abstract methods/classes

In order to give our agents the ability to simulate abstract classes or abilities, we
only need to enforce certain behaviours on our agents, which can be done with
group policies.
To simulate an abstract ability $\varphi$, the agent will have a policy concerning its content that only allows agents to join that do have ability $\varphi$. In the case of abstract classes, this has to be extended to a set of methods. A possible way of implementing abstraction is shown by the following protocol.

\[
\text{receive}_a(\text{addToContent}(b)) \Rightarrow \text{send}_a(b, \text{hasAbility}(\varphi)) \quad (5.2a)
\]
\[
\text{receive}_b(\text{hasAbility}(\varphi)) \land A_b(\varphi) \Rightarrow \text{send}_a(a, \text{hasAbility}(\varphi)) \quad (5.2b)
\]
\[
\text{receive}_a(\text{hasAbility}(\varphi)) \land \neg A_a(\varphi) \Rightarrow \text{send}_a(a, \text{noAbility}(\varphi)) \quad (5.2c)
\]
\[
\text{receive}_a(\text{hasAbility}(\varphi)) \Rightarrow \text{send}_a(b, \text{addToContent}(b)) \quad (5.2d)
\]
\[
\text{receive}_a(\text{noAbility}(\varphi)) \Rightarrow \text{send}_a(b, \text{disconnect}(b)) \quad (5.2e)
\]

It should be clear how formulae (5.2a) to (5.2e) can be extended to deal with sets of abstract methods, or abstract classes. Note that for clarity, we added subscripts to denote the agent sending or receiving (rather than having this information in the predicates being transferred).

Note that this technique can also be used to force agents to adhere to certain interfaces.

### 5.3 Structure and Information

We use the structured agent space not only to represent teams of agents, but also to represent task structures. Because agents can participate in many groups, and groups can have very different and intricate policies, we can use structure to represent, concurrently, many different aspects of a working multi-agent system.

One aspect of multi agent systems that has received much attention in the research community is teams (for example [LCN90, CL91, GK96]). However, grouping agents can be used to represent much more than simply teams, such as physical or task structures, meta-information, or combined abilities.
Examples in Sections 7.1 and 7.2 show how physical structure such as the layout of a building, can be represented. By using simple agents to represent rooms and corridors, agents can not only navigate through buildings, they can also find services that are closest to them. In the Coffee Example, a user sends a request for the coffee robot out, and the room agents forward the request to connected corridors and rooms. In this way, the coffee robot closest to the requesting user will receive the request for coffee and be able to act accordingly.

In Section 7.3, we present the outlines for a travel agent that organises a vacation. Given some initial information such as dates and possible destinations, the agent creates sub-groups that collect together flight information agents, hotels etc. Invited agents can themselves contain agents related to their current task. The resulting structure reflects not only the original task, but also the subtasks that agents have to complete in order to achieve the original task.

The museum example (Section 7.2) also shows how information about visitor agents (such as their preferences) can be represented using the agent space structure. There, visitor agents do not need to keep information as specific agents in the structure keep it, and can be accessed by agents without needing explicit names of agents containing the information.

The above examples only scratch the surface of the possibilities. For example, Microsoft agents could be members of a security-update agent, cars could dynamically be members of agents that convey information about surrounding traffic issues, and so on.

Agents can make use of several different structures simultaneously. It should be clear though that this depends on messages sent being distinguishable by the agents. Agents themselves are not aware of the global structure of the system other than their immediate Content and Context.
5.4 Manipulating the Agent Structure

Given our use of structure, agents will move through the agent space, not just being in a (rather being the) structure. On the most basic level, agents have only very limited possibilities to move about. In fact, “movement” is reduced to adding and removing agents from their Content and Context. However, those primitives can be used as building blocks for protocols that achieve true “movement”. While the programmer can create very elaborate movements commands and protocols, we will cover here just some basic notions, and show how they are implemented.

Generally, we distinguish between actual movement, i.e. an agent moving within the structure, and actions that change the structure but cannot be regarded as movement. Examples of the former would be moving “up” inside a structure, while examples of the latter include the creation and cloning of an agent.

5.4.1 Moving Agents

If structure is used to, for example, represent location, agents that move through the space will have to traverse (part of) the agent structure. We provide some basic constructs that allow the agent to do that.

The most general actions that constitute movement are the addition and removal of agents to/from Content and Context. The system provides built-in side-effects that simply add and remove agents, without any checks to ensure that the connection is mutual.

Figure 5.3 gives a very simple definition for addToContent/2 based on the internal predicates doAddToContent/1, doAddToContext/1, which indiscriminately connects agents. In more complex scenarios, this can be adapted to, for example, allow only certain agents to join the agent.
Building on these defined movement commands, we provide four further actions. Two of them, `moveUp/3` and `goUp/3` (see Figure 5.4(a)) allow the agent to move “upwards” through the structure, whereas `moveInto/3` and `goInto/3` (see Figure 5.4(b)) are for the agent to move “downwards”. The difference between the “move” and “go” actions is that the former is initiated by the agent that is moving, while the latter is initiated by the agent that will serve as new group agent. Figure 5.4 shows the different actions (top row) and the resulting structure (bottom row).

A simple implementation of the `goUp` command is shown in Figure 5.5, and works as follows. If an agent $F$ wants an agent $E$, residing in its content, to move

```plaintext
addToContent: {
    addToContent($SELF,Sender)
    => NEXT doAddToContent(Sender).
    addToContent($SELF,Sender)
    => NEXT send(Sender, addedToContent($SELF,Sender)).
    addedToContent(Sender,$Self)
    => NEXT doAddToContext(Sender). }
```

Figure 5.3. Code block implementing `addToContent`
up such that it will no longer be in $F$’s content but in one of $F$’s context agents (such as $G$), it initiates the goUp command. It takes three arguments, namely the initiating agent ($F$), the agent that is to move ($E$), and the agent ($G$) into whose content $E$ will move. Note that while $G$ should be in $F$’s context, we do no such checks in this example. The rules in Figure 5.5 are part of agent $E$’s behaviour and are fired when it receives the message to move up one level. Upon receiving such a request, $E$ proceeds to (a) send a nested message to $G$ (via $F$) requesting to be added to $G$’s content, and (b) send a message to $F$ requesting to be removed from that one’s content.

```plaintext
/** goUp(Sender, MovingAgent, TargetAgent)
 * movingAgent is the agent making the movement,
 * to the Content of TargetAgent
 */
goUp: {
  goUp(Sender, $Self, Target) => NEXT
  send(Sender, send(Target, addToContent(Target,$Self))).
  goUp(Sender, $Self, Target) => NEXT
  send(Sender, removeFromContent(Sender, $Self)).
}
```

Figure 5.5. Code snippet showing a simple implementation of the goUp action

We provide similar rules to implement the other three movement operators. Again, it is important to note that the given protocols are very rudimentary, and will most certainly have to be extended for more complex scenarios.

### 5.4.2 Manipulating Agents

Another type of rule that we provide with the agent manipulates the agent space, but does not involve movement of agents. Instead, we provide the actions clone, merge, die, and create (see Figure 5.6).
**create** creates and starts a new agent within the content of the original agent. The new agent’s rule base, as well as the predicates that are true at the beginning of time, can be specified.

![Diagram showing create, clone, merge, and die operations](image)

**Figure 5.6.** Manipulating the agent space

**clone** allows an agent to create a clone of itself. Note that we cannot guarantee that the agent’s *Content* and *Context* will be re-created, as agents can refuse to add the clone to their *Content* or *Context*. However, the agent will contain the same rule base, and can get, similar to **create**, unique start parameters. In Figure 5.6(a), the white agent clones itself and the clone restores the *Content/Context* connections to the black agents.

**merge** allows two agents to combine their *Content* and *Context*. While the agents at the moment do not merge their rule base, this action can still be useful to combine agents that provide the same service, especially simple group agents.
5.4 Manipulating the Agent Structure

Merging two agents results in the initiating agent (trying to) receive all Content and Context connections of the second agent, after which the second agents shuts down. Figure 5.6(b) shows how the white agent combines the Content/Context connections within one agent.

die stops an agent gracefully. That is, it not only disconnects from all agents that might populate its Content or Context, and stops executing, but it also tries to connect its Content and Context agents (as shown in Figure 5.6(c)), in order to prevent agents to become disconnected from the structure.

5.4.3 Using Groups

The agent space is obviously dynamic, and agents can be organised, and reorganised, according to the specific scenarios we model or to the actual needs of the agents themselves. We consider here some simple, yet very common, examples of behaviour involving agents belonging to some sort of collections, and we illustrate how to represent these behaviours in our framework.

**Point-to-Point Communication** In order for agent E1 to communicate directly to (and only with) agent E2, E1 may

1. make a new group, called G,
2. add both itself and E2 to G,
3. then broadcast a message, m, within G.

![Diagram](image-url)  
*Figure 5.7. Examples of using structure*
A graphical representation is shown in Figure 5.7(a)

**Filtering**  Agent E is a member of both F and G. When F broadcasts messages to its contents, E will receive these and can pass on selected messages to G. Thus, E acts as an intermediary between F and G, as shown in Figure 5.7(b).

Another, very typical, way of exploiting the structure is to create dedicated group agents that can be used to collect agents for a specific purpose.

**Grouping using Content**

The philosophy behind identifying agents and groups entails that agents will use their *Content* or *Context* to communicate with agents that can contribute to their task (see Figure 5.8).

Thus, at first sight, *Content* provides the perfect means for an agent to enlist others to help it accomplish its goals. As the agent serves as a group for all invited agents (and groups control communication between group members), it has full control over the messages that agents send to each other. It can, for example, choose to not re-send certain messages. Also, as it knows every message that is sent within the group, it receives all messages the agents within its content broadcast to the group.

There are, however, some major drawbacks to this. Recall that one of the advantages of identifying the notions of agent and group was to allow for a variety of policies in groups. If the agent’s only option for enlisting other agents is to invite them to *Content*, it would either mean that the group policy has to be very lax, or the agent would need some way to impose different policies in its *Content* to cater for different tasks. In order to keep communication efficient, and potentially allow communications relating to a certain task to be private
(that is, messages concerning a task would only be sent to agents that have a role in solving it), the agent would need to have exact knowledge of all the tasks that the content agents are supposed to do.

**Agent in Group**

Another method would be to create a new group agent, join its *Content*, and invite all required agents to that group agent, as depicted in Figure 5.9. This would allow the initiating agent to have a strict *Content* policy enforced within the group. Also, groups collect together agents with certain abilities. This imposes a structure on the agent space that corresponds to abilities, in effect representing information about agents, rather than current tasks.

Groups formed in this way can also be re-used. By giving the group agent information about its *Content* (or by the group agent inferring information about its *Content*), it can participate in the agent space as a focus for the type of task for which it was created. Note that the group combines provider and query agents; it is not obvious to participants which agent is which.

![Figure 5.8. Using *Content*](image-url)
However, the initiating agent has less control over such a group, because the original agent is on the same level as the agents joining. If using broadcast communication (by using directed messages to one agent, all of these examples break down), the agent would have little control over which agent performed the task. Note, however, that since the initiating agent is creating the group, it has control over what the overall group behaviour (policy) would be.

**Group in Agent**

A third way of exploiting the structure would be to create a group agent, but instead of joining this group, the agent incorporates it in its *Content*, as shown in Figure 5.10. This way, all provider agents are in the group’s *Content*, while requesting agents would be in the group’s *Context*, providing a clear distinction between provider agents and requesting agents.

In this setting, the requesting agent would not receive any communication occurring within the group, nor would it know about what agents are part of the group, or perform tasks within the group. The only communication between

![Figure 5.9. Using external group](image-url)
requesting agent and group would be the actual request and a subsequent answer.

**Independent Groups**

Lastly, we can totally decouple the agent and the group it needs. In this case, either the group exists already, or the agent creates the group inside one or more of its context agents. The main difference from the previous case is that there, the created group has no context except the creating agent (which can choose to not send any messages from the outside to the group, therefore keeping it private), while here the group is published in (probably) more than one agent, and is therefore more tightly integrated within the agent environment.

**Group Structures**

The presented group structures have different properties, some of which we will highlight here. We will focus on three different properties which are of importance within agent systems.

The amount of control an agent has over a group is obviously of importance.

---

**Figure 5.10.** Using internal group
Control includes the knowledge of which agents are in the group, the ability to remove and add agents, and the ability to privately communicate with an agent.

Another aspect is the visibility of the group to the rest of the system. Basically, groups can be public, or visible to other agents, such that others can make use of the abilities offered, or they can be private to a certain agent (or sets of agents). Last, but not least, the knowledge of communication within the group determines

<table>
<thead>
<tr>
<th>Property</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>low</td>
<td>med</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Visibility</td>
<td>low</td>
<td>med</td>
<td>low</td>
<td>high</td>
</tr>
<tr>
<td>Knowledge</td>
<td>low</td>
<td>high</td>
<td>high</td>
<td>low</td>
</tr>
</tbody>
</table>

**Figure 5.11.** Different properties that groups can have

how involved the agent is in the processes that occur within a group.

Figure 5.11 shows the different structures and their relation to above mentioned properties. Choosing the best structure for a given problem is by no means straightforward, certainly if decisions have to be made at run-time by agents themselves, but even at design time it is not trivial. Using the above properties of control, visibility, and knowledge, however, should make choices easier.

### 5.5 Group Policies

As groups are agents, they can create intricate and complex behaviours with respect to how they move within the agent space. In this section, we will show how even very simple rules allow agents to exhibit very different behaviours when other agents want to join their *Content*. We will do so by describing three different informal varieties of agents, namely trusting, cautious, and demanding agents.

In order to describe the behaviours, we assume the following scenario. Agents are based on the rules $A \varphi \land B \lozenge \varphi \Rightarrow \lozenge \varphi$ (see Section 3.2), and $\neg A \varphi \land B \lozenge \varphi \Rightarrow$
send(A_\varphi),^{2} i.e. agents that have confidence in \varphi will eventually achieve \varphi (if they are able to do so), or ask other agents if they have the ability to bring about \varphi. In our scenario, we assume that the initial agent has the confidence in, but not ability to bring about \varphi. Before agents can exhibit their different behaviour, they will need to establish that there are agents that actually can accomplish the requested task. As we initially assume that B\Diamond \varphi but \neg A\varphi, the first rule is:

B\Diamond \varphi \land \neg A\varphi \Rightarrow send(A_\varphi)

So, A_\varphi is broadcast from the agent. All agents x that have the ability to bring about \varphi are required to answer truthfully, according to the following rule:

receive(A_\varphi) \land A\varphi \Rightarrow send(A_{Self}\varphi)

If an agent does not have the ability to do “\varphi”, it does not answer at all.

Behaviours of Requesting Agents

Assuming that at least one agent responded to the query about ability, we can now distinguish between different types of behaviours.

• A trusting agent, t, upon receiving the message that agent x has the ability to carry out \varphi, will trust that x will bring about \varphi without t’s intervention, and therefore only asserts its confidence in x achieving \varphi within its own (local) knowledge base:

B_t\Diamond \varphi \land \neg A\varphi \land receive(A_x\varphi) \Rightarrow B_t\Diamond \varphi

• A cautious agent, c, upon receiving the message that agent x has ability to do \varphi, will communicate that it believes that agent x will eventually make \varphi true:

B_c\Diamond \varphi \land \neg A\varphi \land receive(A_x\varphi) \Rightarrow send(B_c\Diamond \varphi)

^{2}Note that we omit the receiver set in send and subscripts for readability purposes.
Note that this does not imply that agent $x$ will be forced to actually execute $\varphi$.

- A **demanding** agent, $d$, upon receiving the message that agent $x$ has the ability to do $\varphi$, will request that $x$ actually executes $\varphi$:

$$B_d \Diamond \varphi \land \neg A \varphi \land \text{receive}(A_x \varphi) \Rightarrow \text{send}(\Diamond \varphi)$$

Note that the sends by both the cautious and demanding agents broadcast the message to other agents, not just to $x$. As we will see later, this broadcast of information can be particularly useful in developing team behaviour.

**Behaviours of Responding Agents**

The responding agents, upon receiving the above messages, can choose to either ignore or accept the requests. Again, we describe three distinct types, out of a vast range of possible behaviours. For this purpose we use the following formulae:

- Rule (5.3) says that agent $x$ will bring about $\varphi$ if it gets to know that some other agent $c$ believes that $x$ will do $\varphi$. Rule (5.4) implies that $x$ will honour demands to bring about $\varphi$. (Note that this is a much stronger request.) The last rule (5.5) is in fact formula (3.1) (page 46), which deals with internal commitments.

- A **friendly** agent $x$ will always try to help others, and will therefore not only honour direct requests, but will also execute tasks if other agents are confident that $x$ is executing them. It therefore has rules (5.3), (5.4), and (5.5) in its rule set.
• A neutral agent $x$ is less willing to help agents, and will therefore only have rules (5.4) and (5.5)

• Last but not least, the unfriendly agent never listens to requests or confidences of other agents, and only executes tasks it is confident of itself. Accordingly, it will only contain rule (5.5).

The three examples given here are but a few in a vast range of possibilities — however, we hope that they give a flavour of the possibilities of even simple rules.

5.6 Implementing Groups of METATEM Agents

In this section, we will describe the actual implementation of METATEM groups in a little more detail.

5.6.1 Setting up the Agent Space

In order to program a multi-agent system in METATEM, the programmer needs to define not only the (initial) agents, but also the initial structure in which the agents live. The initial structure of the systems is defined within a so-called environment file. Its precise structure is given in Appendix A; generally, a structure is defined by naming the agents that form the structure, and defining their respective Content and Context. Once all agents have been initialised, and the structure created, the agents are started, each in its own thread.

5.6.2 Enabling Direct Communication

In more complex scenarios, where many agents move constantly through the agent space, communication between distant agents becomes difficult. The two options that the framework allows, namely broadcasting to the whole agent space, or routing the message through the space by means of nested send messages have
both drawbacks. The former floods the system with messages while the latter does not guarantee delivery, as the agent might have moved, thereby invalidating the routing information embedded within the send message. While a further solution is to create a dedicated group agent that contains the two communicating agents, this is not always satisfactory, as it takes a few messages already to establish such a group, and for the agents to enter such a group. We would need to ensure that, while agents are involved in adding themselves to groups, they do not move within the agent space. This might be a viable solution from the logic programming perspective, but in our view it imposes too much of a restriction on agents.

We therefore introduce the *Known* set into the agent system. Every agent has not only two sets that can contain agents (*Content* and *Context*) but a third set that contains references to all agents that have been communicated with (see Figure 5.12). That way, agents that are in contact can *always* communicate. As opposed to *Content* and *Context*, the connections that the *Known* set provides are essentially one-way, though by receiving a message on a direct channel, the sending agent is automatically added to the receiving agent’s *Known* set.

Figure 5.12. Graphical representation of *Known* set
It is important to understand that, while the Known set can be used just as the agent’s Content and Context, it is primarily meant to enable communication within large agent spaces.

Note that agents can only be a member of one of the three sets — if an agent is added to Content, it is removed from the Known set. If it disconnects, it is moved back to the Known set, for future communication. In the implementation, direct is a key word that can be used to address this special set.

5.7 Structural Semantics for Multiple META-TEM Agents

In Section 3.4, we presented the reader with structural semantics for single META-TEM processes. In this section, we will build on the semantics and extend them to describe not only single agents, but multiple agents within a structured agent space.

To do so, we need to cover several aspects of such a system. First of all, we need to define a semantics that allows us to execute agent descriptions concurrently. Secondly, agents need to be able to communicate in some way. While META-TEM is a temporal logic in which predicates have no associated meaning, we will have to introduce the concept of side effects in order to allow agents to manipulate the agent system, for example, send messages or add agents to its Content.

Concurrency we will formalise by interleaving the state transitions of the agents. In order to enable agents to communicate, we adopt a global environment to store messages [NN99]. Each message will be represented by a tuple consisting of the name of the sender, receiver, and the literal that is being communicated. Whenever an agent sends a message, the tuple will be added to the environment,
and removed upon reading. Furthermore, we will have to provide means to record messages that are received, as we have ensure that no messages are lost in case of backtracking.

We will now give some definitions needed to formalise above intuitions.

**Definition 21 (Message)** As mentioned earlier, a message is defined as a tuple $\langle \text{AgentNames}, \text{Literals}, \text{AgentNames} \rangle$ which holds the sender of the message, a literal representing the message, and its receiver. We call the set of messages $\text{MessageSet}$. 

**Definition 22 (Environment)** We define the Environment to be a set of $\text{Messages}$. We will often refer to it as $\text{Environment}$, or $\text{Env}$. 

**Definition 23 (States)** In order to provide a means to store received messages, we extend a state (as defined in Def. 8) to be a tuple $\langle P, Ev, B, DNF, M \rangle$, where $P, B, Ev, \text{ and } DNF$ are defined as in Definition 8, and $M$ of type $\text{MessageSet}$ contains messages that will be incorporated into the next state. 

As we now have multiple agent descriptions, we chose to represent the description of the agent space as a list of agent descriptions, and the corresponding model as a list of histories. The reason for using a list, rather than a tuple, is that agents can create other agents, thereby changing the arity of the tuple. Also, a list allows us to easily define a correspondence between agent definitions on the one hand and agent models on the other hand by using the indices of both lists, the agent description, and the agent models.

**Definition 24 (Agent Space)** We define the agent space as a list of agent descriptions, or an $\text{AgentsList}$. Generally, we will use the variable name $\text{AgentList}$ to refer to the list, and will refer to a specific agent description by $\text{Agent}_m$, where $m$ is the index of the element.
Definition 25 (Multiple Agent Model) While our single agent model consisted of a history, the model for multiple agents, `ModelList`, is defined as a list of histories, or a list of lists of states. We write `HistList` to refer to the model list, and `History_m` to refer to the history at index `m`.

To allow agents to actually manipulate the environment and their `Content` or `Context`, we need to interpret different predicates at each new state that is created, and execute associated effects if there are any. We will now adapt the transition functions introduced in Section 3.4.1 to accommodate the needed changes. We will generally use the same transition names, but add a `m` to the transition arrow (e. g. `→` for single agent case becomes `→_m` for the multi agent case).

Definition 26 (Semantic Function) We define a new semantic function `S_m`, again in terms of a (very) high level transition function `m→_h`: `AgentsList` × `Environment` → `ModelList` for the multi agent case as follows:

\[
S_m(AgentsList, Env) = \begin{cases} 
\text{History} & \text{if } \langle \text{AgentsList}, Env \rangle \xrightarrow{m \text{'h}} \text{History}^m \\
\text{FAIL} & \text{otherwise}
\end{cases}
\]

(5.6)

5.7.1 History Level

The multiple history transition `m→_h` only expresses that executing a number of agents concurrently, a list of histories will be produced. More specifically, this history list will be produced by executing single agent state transitions of the agents. However, because we have to define how agents communicate, we cannot just use the transition function `→_s` but have to define a new transition `m→_s`. More specifically:
Definition 27 (Multiple History Function) We define the multiple histories function $m_h : \text{AgentsList} \times \text{Environment} \rightarrow \text{ModelList}$ as follows, using a global state transition function $m_{gs}$:

$$\langle \text{AgentsList}, \text{Env}, [] \rangle \xrightarrow{m^*_{gs}} \langle \text{AgentsList}', \text{Env}', \text{HistList} \rangle$$

$$\langle \text{AgentsList}, \text{Env} \rangle \xrightarrow{m_h} \text{HistList}$$

where $^*$ denotes the transitive closure of $m_{gs}$.

5.7.2 State Level

The application of a single global step is easily expressed in terms of a transition whereby only one agent makes a state transition. However, we have to take into account that a transition of a single agent not only can alter its own history, but also the environment, if it is sending or receiving messages. Furthermore, agents can create other agents, thereby extending the list of agent descriptions and accordingly the list of histories. Note that if an agent stops its execution, its description as well as history will not disappear from the description. Also note that the choice of which agent’s state transition to choose is made non-deterministically!

More precisely we write:

Definition 28 (Global State Transition) A global state transition $m_{gs}$:

\text{AgentsList} \times \text{MessageSet} \times \text{ModelList} \rightarrow \text{AgentsList} \times \text{MessageSet} \times \text{ModelList}

is defined in terms of a transition $m_s$ of a single agent as follows. Given some (non-deterministically chosen) natural number $i, 1 \leq i \leq |AList|$

$$\langle \text{Agent}_i, \text{Env}, \text{History}_i \rangle \xrightarrow{m_s} [\text{[Agent}'_i\text{][AList}^+, \text{Env}', [\text{History}'_i\text{][HList}^+]])$$

$$\text{AList}' = \text{AList}[\text{Agent}_i/\text{Agent}'_i] + \text{AList}^+$$

$$\text{HList}' = \text{HList}[\text{History}_i/\text{History}'_i] + \text{HList}^+$$

$$\langle \text{AList}, \text{Env}, \text{HList} \rangle \xrightarrow{m_{gs}} \langle \text{AList}', \text{Env}', \text{HList}' \rangle$$

(5.8)
Each state transition of some agent $i$ not only modifies its own history $i$, but potentially alters the environment (by sending or receiving messages) or creates new agents that have to be added to the global list of agents. Generally, however, the lists $AList^+$ and its associated history list $HList^+$ will be empty. If $AList^+$ does contain newly created agents, their associated histories will be empty.

### Single Steps
Recall that the state transition $\rightarrow_s$ (Definition 3.7) of a single agent consists of two steps, namely the interpretation of the agent’s rule base, and the actual choice of a successor state. We will have to adapt this to allow agents to interact. Intuitively, the agent has to do the following: First, it receives any messages and incorporates them into its set of satisfied literals. It then interprets the rule base, expanding the current state to contain possible configurations for a next state, as well as computing current beliefs. If the state is consistent with respect to the beliefs, we execute any side effects, and finally (if the side effects do not yield an inconsistency) choose a successor state. In case of an inconsistency, we move any received messages to the preceding state and choose again. The last step is needed in order to ensure that no messages are lost.

We handle messages within the choice function $\rightarrow_{c}$, while the side effects are dealt with on the state level.

Formally, the state transition for agents in a multi agent environment is defined in terms of several other functions, namely

- $\rightarrow_{r}^\dagger$, defined in Section 3.4.4,
- $\rightarrow_{b}$, defined in Section 3.4.6,
- $\rightarrow_{ex}$, which deals with side effects (see below), and
- $\rightarrow_{m}$, which chooses the next state (also below).

### The Choice Function
As mentioned earlier, we have to adapt the choice function of the single agent case (Def. 14) to deal with receiving messages. For
convenience we repeat the transition rules of the function \( \rightarrow_c \) here:

\[
S = \langle P, Ev, B, [P'|\text{DNF}^+] \rangle, \\
S' = \langle P', \text{merge}(Ev, P'), [], [], [] \rangle, \\
S'' = \langle P, Ev, B, \text{DNF}^+ \rangle \\
\frac{[S|\text{History}] \rightarrow_c [S', S''|\text{History}]}{\text{if } \text{DNF}_S \neq []} \tag{3.8a}
\]

\[
\text{History} \rightarrow_c \text{History}' \\
\frac{[S|\text{History}] \rightarrow_c \text{History}'}{\text{if } \text{DNF}_S = []} \tag{3.8b}
\]

\[
[S] \rightarrow_c [] \\
\frac{\text{if } \text{DNF}_S = []}{\text{ if } \text{DNF}_S = []} \tag{3.8c}
\]

We have to consider the actual receipt of messages, and we need to ensure that no messages are lost in case of back tracking. Receiving messages consists of collecting messages addressed to the agent from the environment, removing them from the environment, and adding them (wrapped within a received predicate) to the new state. In case of back tracking, the messages from the current state need to be added to the preceding state before removing it, so that no messages are lost. More formally, we define the choice function as follows:

**Definition 29 (Choice Function for Multiple Agents)** Given a function \( \rightarrow_{rd} \), which reads messages from the environment, we define the function \( m \rightarrow_c \):

\[
\text{AgentName} \times \text{Model} \times \text{MessageSet} \rightarrow \text{AgentName} \times \text{Model} \times \text{MessageSet}
\]

from History and Environment to History and Environment. Note that for readability we drop the agent name as an element of the tuple, as it is merely needed to allow the agent to retrieve messages sent to it. We therefore define the following rules:
5.7 Structural Semantics for Multiple METATEM Agents

\[
\langle AName, Env \rangle \rightarrow_{rd} \langle Env', Mes \rangle
\]

\[
S = \langle P, Ev, Ev^+, B, [P']|DNF^+, Mes' \rangle,
\]

\[
S' = \langle (P' \cup Mes \cup Mes'), merge(Ev, Ev^+, P'), [], [], [] \rangle,
\]

\[
S'' = \langle P, Ev, Ev^+, B, DNF^+, (Mes \cup Mes') \rangle
\]

\[
\langle [S|History], Env \rangle \xrightarrow{m_c} \langle [S', S''|History], Env' \rangle \quad \text{if } DNF_S \neq []
\]

\[
Mes'' = Mes_S \cup Mes_S'
\]

\[
S'' = S'[Mes'/Mes'']
\]

\[
\langle [S''|History], Env \rangle \xrightarrow{m_c} \langle History', Envo' \rangle \quad \text{if } DNF_S = []
\]

\[
\langle [S, S'|History], Env \rangle \xrightarrow{m_c} \langle [], Env \rangle \quad \text{if } DNF_{S'} = []
\]

The transition function \( \rightarrow_{rd} \): AgentName \( \times \) MessageSet \( \rightarrow \) MessageSet \( \times \) Literals extracts all messages \( m = \langle \text{sender, message, receiver} \rangle \) that are sent to the agent, and creates a set of predicates of the form receive(message). We leave it to the reader to provide a formal definition thereof.

As with the choice function for the single agent case (Def. 14), only Equation (5.9a) actually expands the history. Note that Equation (5.9b) ensures that received messages are carried backwards in case of back-tracking.

We are now ready to define a state transition for the multiple agent setting.

**Definition 30 (State Transition for Multiple Agents)** The function \( \xrightarrow{m_s} \): Agent \( \times \) MessageSet \( \times \) Model \( \rightarrow \) AgentsList \( \times \) MessageSet \( \times \) ModelList we define as follows, again using subscripts to refer to elements of named tuples:
Groups of METATEM Agents

\[
\langle Ag, S \rangle \xrightarrow{r} S',
\]

\[
\langle Ag, [S'|Hist], Env \rangle \xrightarrow{ex} ([Ag'|Ags^+], [[S''|Hist]|HL^+], Env'),
\]

\[
([S''|Hist], Env') \xrightarrow{m} (Hist', Env''')
\]

\[
\text{Ags} = [Ag'|Ags^+], \quad \text{HList} = \text{Hist'} + \text{HL}^+
\]

\[
\langle Ag, Env, [S|Hist] \rangle \xrightarrow{s} \langle Ags, Env'', \text{HList} \rangle
\]

if \( S' \) consistent

\[
\text{(w.r.t. B)}
\]

and \( \xrightarrow{ex} \text{ok} \)

\[ (5.10a) \]

\[
\langle History, Env \rangle \xrightarrow{m} (\text{History}', Env')
\]

\[
\langle Ag, Env, [S|Hist] \rangle \xrightarrow{s} ([Ag], Env', [\text{History}'])
\]

otherwise

\[ (5.10b) \]

Side Effects For ease of writing, we will abuse notation somewhat and define \( \xrightarrow{ex} \) as the application of all transition functions \( \xrightarrow{\text{name}}_{ex}: \text{Agent} \times \text{State} \times \text{MessageSet} \rightarrow \text{AgentList} \times \text{HistList} \times \text{MessageSet} \) for which holds that the literal \( \text{name} \) is satisfied in the current state (\( S' \) in Equations (5.10)). This allows us to easily extend the semantics to incorporate literals with new side effects.

In a further abuse of notation, we distinguish between two types of side effects. The most common one modifies the agent’s state or the environment, and includes the sending of messages and creation of new agents. However, we also allow side effects that do not modify anything but rather evaluate to either true or false. In the former case, the execution continues, while in the latter case, we stop expanding the state and instead backtrack, i.e. the state will be considered inconsistent. We can use this type of side effect to implement, for example, equality checks. We will now present the semantics for a few predicates with side effects.
5.7 Structural Semantics for Multiple MetateM Agents

send(SetExpression,Message) sends messages to the environment. Formally, sending messages is described by adding Message elements to the environment. Before we can do so however, we need to provide means to interpret set expressions.

Definition 31 (Interpreting Set Expressions) To deal with set expressions we define a transition function $\rightarrow_{se}: \text{SetExpression} \times \text{Agent} \rightarrow \mathcal{P}^\text{AgentName}$ from set expressions to a set of agent names as follows. For readability, we again drop the agent argument as it is only needed to provide access to the agent’s Content and Context:

\[
\begin{align*}
SE_1 \rightarrow_{se} ANames_1 & \quad SE_2 \rightarrow_{se} ANames_2 \\
SE_1 \cup SE_2 & \rightarrow_{se} ANames_1 \cup ANames_2 \\
SE_1 \cap SE_2 & \rightarrow_{se} ANames_1 \cap ANames_2 \\
SE_1 \setminus SE_2 & \rightarrow_{se} ANames_1 \setminus ANames_2
\end{align*}
\]

\[
\begin{align*}
\text{content} & \rightarrow_{se} content_{Agent} \\
\text{context} & \rightarrow_{se} context_{Agent}
\end{align*}
\]

\[
SE \rightarrow_{se} SE \quad \text{if } SE \in \mathcal{P}^\text{AgentNames}
\]

We now can define the semantics of the predicate send as follows. For each predicate of the form send(set, message), we first interpret the set expression set to get the agents that should receive the message, and then add a message tuple for each of the receiving agents to the environment. Finally, as we do not allow the agent to backtrack once it has sent messages, we remove the current history.
More formally:

\[ \forall p \in P_S : p = \text{send}(\text{set}, \text{mes}) : \text{set} \rightarrow_{se} \text{Agentset} \, \text{and} \]

\[ \forall a \in \text{Agentset} : \text{Env}' = \text{Env} \cup \{ \langle \text{AName}_{\text{Agent}}, \text{mes}, a \rangle \} \]

\[ \langle \text{Agent}, [S|\text{Hist}], \text{Env} \rangle \xrightarrow{\text{send}} \langle [\text{Agent}], [S], \text{Env}' \rangle \]

(5.12)

\text{doAddToContent}(\text{AgentName}) \quad \text{simply modifies the Content variable of the agent description to include the agent. Formally:}

\[ \forall p \in P_S : p = \text{doAddToContent}(a) : \]

\[ \text{Agent} = \langle \text{Name}, \text{Content}, \text{Context}, \text{Rules}, \text{BeliefDepth} \rangle \]

\[ \text{Agent}' = \langle \text{Name}, \text{Content} \cup \{ a \}, \text{Context}, \text{Rules}, \text{BeliefDepth} \rangle \]

\[ \langle \text{Agent}, [S|\text{Hist}], \text{Env} \rangle \xrightarrow{\text{doAddToContent}} \text{ex} \langle [\text{Agent}'], [[S]|\text{Hist}], \text{Env} \rangle \]

(5.13)

doAddToContext works in a similar fashion.

\text{doRemoveFromContent} \quad \text{and doRemoveFromContext again are very similar to doAddToContent, but remove agents from the sets. If they are not elements of the list, we ignore the predicate.}

\text{doCreateAgent} \quad \text{adds a new agent to the set of agents.}

The information needed to create a new agent is the agent name, a rule set and a set of literals that are true at the start of the agent’s execution. Note that, the set of starting literals is transformed into a set of start rules that make the literals true at the beginning of time, using the function \text{makeStart}. Also, the Content and Context of the new agent are both empty at first, so the creating agent has to ensure that rules are set in place to create initial communications:

\[ \forall p \in P_S : p = \text{doCreateAgent}(\text{Name}, \text{Rules}, \text{Literals}) : \]

\[ \text{Rules}' = \text{makeStart}(\text{Rules}, \text{Literals}) \]

\[ \text{Agent}' = \langle \text{Name}, \{ \}, \{ \}, \text{Rules}', \text{BeliefDepth}_{\text{Agent}} \rangle \]

\[ \langle \text{Agent}, [S|\text{Hist}], \text{Env} \rangle \xrightarrow{\text{doCreateAgent}} \text{ex} \langle [\text{Agent}, \text{Agent}'], [[S]|\text{Hist}], [], \text{Env} \rangle \]

(5.14)
doCloneAgent clones an agent.

In order to clone an agent, we need a new name, as well as a set of predicates that will be made true initially, in order to allow the agent to for example initiate Content and Context connections needed to complete the cloning process. Note that cloning is very similar to the creation of a new agent. Furthermore, we assume here that the new agent name is unique.

\[ \forall p \in P_S : p = \text{doCloneAgent}(\text{Name}, \text{Literals}) : \]

\[ \text{Rules}' = \text{makeStart}(\text{Rules}_\text{Agent}, \text{Literals}) \]

\[ \text{Agent}' = \langle \text{Name}, \{\}, \{\}, \text{Rules}', \text{BeliefDepth}_\text{Agent} \rangle \]

\[ \langle \text{Agent}, [S|\text{Hist}], \text{Env} \rangle \xrightarrow{\text{doCloneAgent}} \langle \langle \text{Agent}, \text{Agent}' \rangle, [[S|\text{Hist}], []], \text{Env} \rangle \]  

(5.15)
Part III

Applications
Chapter 6

Single Agent Examples

In this chapter we will give some examples of Structured METATEM programs. With these examples we hope to be able to show to the reader that Structured METATEM can be used to model (at least from an academic point of view) interesting problems, and also to convey how actual programs might look in Structured METATEM.

The following sections will highlight the workings of different elements of the system, such as simple backtracking, the use of bounded belief, and deliberation.

6.1 Backtracking

The running example of Chapter 3 showed how the system employs backtracking whenever it encounters an inconsistent state. Figure 6.1 repeats the program for convenience.

Structured METATEM executes the program as follows. At the initial state, only \( x() \) is true. this forces the next state to satisfy either \( a() \), \( b() \), or both. As a first choice we try to make as few literals true as possible, so the algorithm chooses \( b() \) initially. However, as this choice leads to a state that only provides inconsistent successor states, we revert to state one and choose to make \( a() \) true,
Name bt;
Debug true;
statedebug true;
ability beep:Beep;

single:
  START => x().
  x() => NEXT a(); b().
  b() => NEXT d().
    => NEXT not(d()).
  a() => NEXT end().

Figure 6.1. Example Program to show backtracking

which leads to a consistent state.

Figure 6.3 shows an (abbreviated) execution trace as output by the implementation with statedebug set to true. In order to understand this figure, we should note the following. At each state, the system prints out the CNF of literals that must be made true at the next moment in time, which together with eventualities (becomes with []) are transformed into a (sorted) DNF representation. It then prints the positive and negative literals (using[],[...]) and the rules that fire based on those literals. Also note that rules that fire by default appear in the CNF prefixed by True, which is internally used to distinguish rules that

Figure 6.2. Graphical representation of execution trace
fire by default from rules that fire by virtue of literals that are forced to hold in that state. Figure 6.2 shows graphically how the system tries different states, backtracking whenever an inconsistency is found.

As Figure 6.3 shows, the algorithm tries all possible combinations of literals in state 2 in order to create a consistent next state, but fails. It therefore discards state 2 and makes the next choice in state 1, i.e. $a()$. This yields a consistent state which then is repeated until the system detects a repeating state, upon which it stops and waits for incoming messages (which will not occur in this setting).

6.2 Bounded Belief

Next, we show how belief bounds can be used to change the agent’s behaviour. We borrow an example used in [FG00], though present a somewhat simplified version here.

6.2.1 Setting

In this example, we model a football agent. The agent is near the opponent’s goal, and has to decide whether to try and score, or to pass the ball to a mate who can then score. Figure 6.4 shows the distribution of players. (Note that in this setting, we ignore the off-side rule. The concerned reader can assume another player, $d$, standing before agent $b$, in a position such that it cannot prevent agent $b$ from scoring). Agent $a$, which we are modelling here, has two possible courses of action. Either it can do $\alpha$ and try and score, which has a very low probability of succeeding, or it can go for $\beta$, which means passing on to $b$ ($\beta_1$) which then scores ($\beta_2$). The agent believes that if it shoots, it will miss, and that if it passes, it (that is, the team) will score. Figure 6.5 shows the source code of the
Figure 6.3. Execution trace of program shown in Fig. 6.1
agent. The rules specify that the agent can either shoot or pass. From shooting follows that the agent misses, whereas passing leads to scoring. Further rules describe the belief, which states that the agent believes that it will miss if it tries to shoot. Note further the trick used to program \textbf{False}, by using two rules that force inconsistent predicates in the next state (namely \texttt{bad(X)} and \texttt{not(bad(X))}). This inconsistency is used to force that (in this simple setting) belief contexts which make \texttt{not(score(X))} true, fail, because there are no choice points in the belief contexts.

We can now have two scenarios. In the first one, the agent has plenty of time to “think” about its next action. Its belief depth is therefore set to 100. In the second scenario, the agent has very little time to think, and therefore sets the depth of belief exploration to 0. The agent’s behaviour changes as follows.

In the first scenario, after we start the game (\texttt{play} becomes true), the agent can either shoot or pass. Assume it chooses to shoot. Then, at the next moment in time, it \texttt{shoot} becomes true, and with it the belief that it cannot score. Furthermore, its goal is to score, which it will not achieve now, as \texttt{not(score)} is
true. As it encounters a inconsistency in its belief state, the state that triggered
the belief exploration (which had score true), becomes inconsistent too, and the
agent backtracks to another choice, which is to pass the ball. It now not only
arrives at a consistent state, it also believes that it (the team) will score.

In the second scenario, the belief states are not explored, and therefore the
agent never arrives at the conclusion that it must not shoot in order to score. It
therefore shoots, and loses the game. Figures 6.7 and 6.6 show the (truncated) output of the respective executions.

```
football(0)(0)(0) next Choice 0 at 1(2,0): [play(1)]
football(0)(0)(0) next Choice 0 at 2(2,1): [shoot(1)]
football(0)(0)(0) next Choice 0 at 3(2,2): [miss(1)]
football(0)(0)(0) next Choice 0 at 3(1,0): [miss(1), donot(1)]
football(0)(0)(0) next Choice 0 at 3(1,1): [print(I LOST),
   bad(1), not(bad(1)), stop()]
football(0)(0)(0) next Choice 1 at 2(2,1): [pass(1)]
football(0)(0)(0) next Choice 0 at 3(2,2): [score(1)]
football(0)(0)(0) next Choice 0 at 3(1,0): [score(1)]
football(0)(0)(0) next Choice 0 at 3(1,1): [stop(), print(I WON)]
```

**Figure 6.6. Output of run with belief depth 2**

```
football(0)(0)(0) next Choice 0 at 4(2,3): [stop(), print(I WON)]
```

```
football(0)(0)(0) next Choice 0 at 1(0,0): [play(1)]
football(0)(0)(0) next Choice 0 at 2(0,1): [shoot(1)]
football(0)(0)(0) next Choice 0 at 3(0,2): [miss(1)]
BELIEF DEPTH REACHED or no beliefs
football(0)(0)(0) next Choice 0 at 4(0,3): [print(I LOST), stop()]
BELIEF DEPTH REACHED or no beliefs
football(0)(0)(0): I LOST
```

**Figure 6.7. Output of run with belief depth 0**

6.2.2 Discussion

The football example highlights different features of METATEM. On a technical note, the program shown in Figure 6.5 employs the use of variables to ensure that rules only fire when they should. If we remove the variable and allow all rules to become true, the systems computational requirements become much higher (it takes about 90 minutes to compute, rather than seconds (on the same machine)).
This stems from the fact that the system has many possible combinations of literals that it can make true.

Furthermore, the example shows how the agent’s behaviour can be determined by the amount of reasoning about their beliefs. While we agree that the example is a simple one, it shows how agents can reason adaptively based on the allowed depth of belief exploration. Using the internal predicate \texttt{setDepth/1}, agents can dynamically change the amount of processing they are willing to spend on exploring belief contexts. It might be clear however that while it is easy to change the depth, it is by no means obvious how agents should choose an appropriate depth.

### 6.3 Deliberation

In this section, we will show how agents can deliberate using the built-in predicate \texttt{prefer/2}. As described in Section 3.2.1, \texttt{prefer/2} re-orders the eventualities such that the system tries to satisfy the first argument before the second one as follows. Recall that eventualities are ordered by the time of their appearance, such that the oldest outstanding eventuality is attempted first. \texttt{prefer} does nothing if the list of eventualities is consistent with the order requested by the predicate. Otherwise, it moves the first argument directly before the second one. For example, if the list of eventualities were \([a,b,c,d]\), and \texttt{prefer}(d,a) would become true, the eventualities list would become \([d,a,b,c]\).

From this example it should be clear that the usage of this meta-predicate can cause the loss of completeness. While this is true of other approaches of adding deliberative capabilities to \textsc{MetateM} [Fis97a], our approach has the advantage that deliberation, even though it is implemented using meta-rules, can be programmed and therefore manipulated from within the agent description. Fisher assumed a function outside the agent definition that reordered the eventualities
during each execution cycle, which not only forces the programmer to write the deliberation part in another programming language, but also independent of any events that happen during agent execution.

### 6.3.1 Setting

In order to show how prefer works, we run here a small example, the source code of which can be found in Figure 6.8. The program describes a simple schema for going to a restaurant. It states that going to a restaurant entails that one has to eventually *order*, *eat*, and *pay*. However, depending on the restaurant, the order in which those actions take place is different. In our example, the cheap and cheerful “Kimos” requires customers to pay before eating, while more distinguished establishments usually allow customers to eat before presenting a bill. Figure 6.9 shows that, upon execution, the agent first “goes” to Kimos and orders, pays, and eats, after which it visits a fancy restaurant, where it orders, eats, and pays. Note that if we had not put several steps between the “actions” of eating, ordering, and paying on the one hand, and the actual printing of these actions to the screen on the other hand, the program outputs “ordering” many times, as the systems backtracks to satisfy the constraints.

Furthermore, we allow only one action (of ordering, eating, and paying) to happen at one moment in time. This is achieved by forcing an inconsistency whenever more than one of the actions are attempted at the same moment in time.

### 6.3.2 Discussion

We have mentioned at the beginning of this section that prefer can render agent programs incomplete. In our opinion however, the benefits outweigh the drawbacks. Even in the very simple example given, it is not immediately obvious
name deliberate;
ability print: Agent/Print;

think: {
  start => begin().

  begin() => NEXT go_restaurant("Kimos");
  go_restaurant(X) => sometime order(X).
  go_restaurant(X) => sometime eat(X).
  go_restaurant(X) => sometime pay(X).

  order(X), eat(X) => next doNot(X).
  order(X), pay(X) => next doNot(X).
  eat(X), pay(X) => next doNot(X).
  eat(X), order(X), pay(X) => next doNot(X).

  doNot(X) => NEXT this(X).
  doNot(X) => NEXT not(this(X)).

  => next prefer(order("kimos"), eat("kimos"));
  => next prefer(order("kimos"), pay("kimos"));
  => next prefer(pay("kimos"), eat("kimos"));

  => next prefer(order("fancy"), eat("fancy"));
  => next prefer(order("fancy"), pay("fancy"));
  => next prefer(eat("fancy"), pay("fancy"));

  order(X) => NEXT q1(X).
  q1(X) => NEXT q11(X).
  q11(X) => NEXT print(order(X)).
  pay(X) => next q2(X).
  q2(X) => NEXT q22(X).
  q22(X) => NEXT print(pay(X)).
  eat(X) => next q3(X).
  q3(X) => NEXT q33(X).
  q33(X) => NEXT print(eat(X)).

  eat("kimos") => NEXT go_restaurant("fancy").
}

Figure 6.8. Example code for deliberation
6.3 Deliberation

\[
\begin{align*}
&\text{deliberate}(0)(0)(0): \text{order}(\text{kimos}) \\
&\text{deliberate}(0)(0)(0): \text{pay}(\text{kimos}) \\
&\text{deliberate}(0)(0)(0): \text{eat}(\text{kimos}) \\
&\text{deliberate}(0)(0)(0): \text{order}(\text{fancy}) \\
&\text{deliberate}(0)(0)(0): \text{eat}(\text{fancy}) \\
&\text{deliberate}(0)(0)(0): \text{pay}(\text{fancy})
\end{align*}
\]

**Figure 6.9.** Example output of restaurant agent

how we could achieve the result of forcing eventualities to be executed in different orders without the usage of a different ordering function (which would be outside of the agent program). (We concur that given the simplicity of the program, a set of next rules can be devised. However, as we can see in Section 7.2, when dealing with (multiple) messages, it is much less obvious, how to do so). Furthermore, the preference network can evolve over time and we could, in principle, check whether the evolving preference network retains completeness.

As described in Section 3.2.1, prefer/2 re-orders eventualities by moving the preferred predicate immediately before the less preferred one (if they are not already in order). It should be noted here that this is by no means perfect, as the order of eventualities might depend on the order in which prefer-rules are executed.

It is our understanding that prefer/2 should be used in situations where it is reasonably well understood in which context eventualities appear. By this we mean that (other than in the example), preferences should appear only when they actually make sense. Furthermore, it should be understood which other eventualities might be outstanding. In the example, as well as in the example shown in Section 7.2, we order a set of eventualities that become true at one moment in time — something that, in our experience, happens on a regular basis, and where it is usually important to enforce certain orderings.
Chapter 7

Multiple Agent Examples

In this chapter we consider a number of different examples involving multiple agents, showing how each can be programmed in our framework.

7.1 Coffee Example

7.1.1 Setting

Suppose we have an office building, with an arbitrary number of rooms, attached to corridors, which again are connected by crossroads, and finally floors. Suppose, furthermore, that we want to create a system that lets users instruct a coffee agent to bring coffee to their desks. We represent each room and corridor by an agent. Each corridor agent contains the room agents that it connects to, and is itself contained by crossroad agents. Note that room agents can be members of more than one corridor agent.

The system is initialised by defining an initial structure using a simple language. To define two corridors with 3 rooms each, we could use a definition such as in Figure 7.2. The first part defines the agents and their definition files, while the second part defines the relation between the agents.
A user now sends a request for coffee. Each room agent has a definition file similar to the one depicted in Figure 7.3, and the robot has a rule base as depicted in Figure 7.4. Also, for brevity, we use “,” to encode a logical “and” operator. In our actual implementation, logical “and” needs to be implemented by using separate rules for each conjunct. The rules for room agents work as follows. Upon receiving a request for coffee, the room agent forwards the request \texttt{needCoffee/3} to its \textit{Content} and \textit{Context} (see rule block \texttt{forwardrequest} in Figure 7.3). Note

\begin{verbatim}
agent room1: room1.agent;
agent room1: room2.agent;
agent room1: room3.agent;
agent corr1: corr1.agent;
...
room1: {context: corr1;
    content: robot1; }
coor1: { content: room1, room2, room3}
...
\end{verbatim}

\textbf{Figure 7.2.} An example of an environment definition
7.1 Coffee Example

```prolog
name room1; debug false;

forwardrequest: {
  receive(needCoffee(User,ID)), not(sentRequest(ID)) => NEXT
  send({content UNION context}, needCoffee($Self,needCoffee(User, ID),ID)), sentRequest(ID).
  receive(needCoffee(Room,User,ID)), not(sentRequest(ID)) => NEXT
  send({content UNION context},
       needCoffee($Self,needCoffee(Room, User,ID),
                 ID)),
       sentRequest(ID).
}

remember: {
  sentRequest(ID), not(receive(delivering(ID))) => NEXT
  sentRequest(ID).
}
```

Figure 7.3. An example file for a room agent

that while the original message received had only two arguments (the name of the user and an argument that identifies the request), the forwarded message has three arguments. The first one is the room’s own name, the second is the original received message, and the last is the identifier for the message. Another rule deals with a received needCoffee/3 message, i.e. a message that was received from another room, rather than a user.

Nesting needCoffee predicates allows the coffee robot to re-trace the path of the message and, by following the route of the message, it can deliver the coffee, by traversing through the needCoffee predicate as follows. Upon receiving a needCoffee request, the agent moves to the room that has forwarded the message to the room the robot is in at that moment. Once it has completed the movement (addedToContent is received), it repeats the procedure with the truncated needCoffee predicate. The actual information is kept “alive” during intermittent steps in time in the coffee robot using the predicate nextMove, which is kept true at each next moment until addedToContent indicates that the movement has been successful. Once a two argument needCoffee is reached, the robot
has found the user. Appendix B.2.5 shows the exchange of messages between user, rooms, and robot. Note especially how the message gets larger due to the nesting of needCoffee predicates.

7.1.2 Discussion

The above example is obviously a simplification. For example, we do not deal with the situation where more than one coffee agent exists. We also assume that there will not be more than one request at a time. However, one can see that neither the robot nor the rooms have any explicit knowledge about their location. In fact, the latter do not even know the number of rooms / corridors they are connected to, yet the robot will move through the space without difficulty. Messages travelling through the agent space will automatically plot the shortest route (in terms of number of rooms, rather than actual distance) for the agent.

```plaintext
name coffeerobot; debug false;

receiving_request : {  
  receive(needCoffee(Room, needCoffee(NewRoom, User, ID), ID)) => NEXT  
    send(Room, removeFromContent(Room, $Self)),  
    send(NewRoom, addToContent(NewRoom, $Self,)).  
  receive(needCoffee(Room, User, ID)) => NEXT  
    nextMove(User).  
}  

keep_walking: {  
  nextMove(User), not(receive(addToContent(Room, $Self))) => NEXT  
    nextMove(User).  
  nextMove(needCoffee(Room, User, ID)) => NEXT  
    receive(needCoffee(Room, User, ID)).  
  nextMove(needCoffee(User, ID)) => NEXT  
    print(broughtCoffee(User, ID)).  
}  

Figure 7.4. An example file for a coffee robot
```
7.2 Museum Example

This example is a rather complex one, bringing together agent deliberation, agent representation of both physical and virtual spaces, and agent migration.

7.2.1 Setting

In work on the PEACH project\(^1\) [SZ02] the concept of “active museum” is being investigated. This is a form of *active environment* [McC01], and can be seen as a large scale multi-user, multi-media, multi-modal system. In the case of PEACH, museum visitors are provided (either on demand or pro-actively, depending on the context) with information about exhibits they may see within the museum. This information may be drawn from a variety of information sources and media types (museum server, online remote servers, etc.), and presented to the visitors by a variety of clients (for example, hand-held devices such PDAs, kiosks, wall screens, and so on).

Generally speaking, active environments have some characteristics that make them substantially different from traditional computing and HCIs. For instance, multiple users may be in a single place, interacting with different applications simultaneously. The set of users changes dynamically over time. Users are unaware (and uninterested) that the environment is formed of many distributed components. Therefore, they interact with the environment as if it were a single, monolithic system. However, services are provided by a variable set of components that join and leave the environment on mobile devices or that may be running anywhere else. Services provided by these components can (partially) overlap; therefore, they need to coordinate in order to decide, for instance, who provides a specific service, and how, in a specific context.

In our reference scenario, an active museum, users’ positions and resource

\(^1\)http://peach.itc.it
availability may impose constraints on the generation and display of information; resources may rapidly change over time, while users move around. In the background, user modelling agents silently record histories of user interactions and build profiles by observing their behaviour; their goal is to customise presentations, avoiding repetitions or inappropriate content. Furthermore, one long-term objective of the PEACH project is supporting groups of visitors, such as families or classes of children, by providing tools that allow the sharing of experience and improve learning. All these requisites imply intensive communication among the software collaborating to provide services well beyond current communication architectures and service composition techniques. The objective here is to create a highly distributed and dynamic environment, where the number of components capable of providing services continuously varies.

The implementation of an active museum that has been provided in PEACH relies on the ability of sending messages to roles, rather than to individual components, and overhearing conversations happening among any set of components of the system. This enables the aggregation of service-providing agents into teams that have been called implicit organisations [BKMR04, BMRL03]. In turn, this enables context-sensitive behaviour to be built into objects embedded in the environment, freeing high-level applications (concerned, for instance, with supporting knowledge sharing within a group) from issues concerning service composition and delivery in a specific environment. The implementation of this idea is based on a form of group communication called channelled multicast [BDN02], which is supported by an experimental communication infrastructure, called LoudVoice. LoudVoice supports the creation of channels on-the-fly; messages sent on a channel are received by all agents tuned into it.
**Metatem in the Museum**

Using the active museum as backdrop, we now show how the dynamic grouping structure and executable logic within the system can be exploited to represent different aspects of the scenario, and how we can design different layers and easily incorporate them into one system.

In our particular example, we chose to use two distinct layers. On the one hand, we use agents to represent the physical space, that is, rooms, exhibits, and visitors; on the other hand we wish to represent an organisational structure that will allow a visitor to receive preferences on the exhibits available in a room.

Using those two structures allows us to exploit the information they contain. The physical structure can be used to keep track of agent’ positions, compute routes (on the fly), find nearby other agents etc. The organisational structure allows agents to receive appropriate suggestions or information, find and communicate with agents with the same interests, profile visitors and so forth.

The first grouping structure (as depicted in Figure 7.5) represents the physical layout of the museum. In our simplified example, museum M has two rooms (R1 and R2), with three exhibits in each room (Ex1...Ex6). Each room also contains a visitor (V1 in R1 and V2 in R2).

A separate organisational structure (Figure 7.6) shows three *interest groups*, the Artist Group (AG), ColourBlind (CB), and Time Group (TG). Visitor1 (V1)
is member of both Artist Group and ColourBlind Group while Visitor2 (V2) is only member of Time Group.

The function of the interest groups is to provide visitors with an optimal path through the room, given the preferences of the visitor. In our example, V1 is interested in artists, so the system suggests a certain order to the exhibits. The group CB tags exhibits that primarily consist of hard to discern shades of red and green (for the sake of argument). It therefore recommends participants not to look at certain exhibits at all.

Given just the physical structure, the system should show the following behaviour. Upon entering a room (joining the content of R), a visitor sends a message to its context, informing it that it is looking around within R. R in turn asks its content what exhibits are available, and forwards the answers to the visitor.

The METATEM Rules needed to accomplish the above are rather straightforward. Figure 7.7 shows the rules needed for V to “look around” and remember the exhibits it can see, as well as the rules R uses to send appropriate answers. The above rules are enough to allow our visitor agent to guide a visitor through a museum — given it has information about the different exhibits, or receives them from the room. However, each and every visitor would be directed towards the same sequence of exhibits.

We now add our organisational layer. As mentioned before, we wish to expand the system by allowing for interest-based guidance through exhibits, possibly excluding exhibits from the list. Figure 7.6 gives the (simple) structure of our

![Figure 7.6. Organisational Structure of Museum Example](image-url)
VISITOR
exhibits: {
    addedToContent(Room,$Self) => NEXT lookAround(Room).
    addedToContent(Room,$Self), canSee($Self,Room1,Exhibit)
    => NEXT seen(Exhibit).
    lookAround(Room) => NEXT send(context, looking($Self,Room)).
    receive(canSee($Self,Room,Exhibit))
    => NEXT canSee($Self,Room,Exhibit).
    canSee($Self,Room,Exhibit), not(seen(Exhibit))
    => NEXT canSee($Self,Room,Exhibit). }

ROOM
exhibits: {
    receive(looking(Visitor,$Self))
    => NEXT send(content,whatExhibit($Self,Visitor)).
    receive(exhibit(Exhibit,$Self,Visitor))
    => NEXT send(Visitor,canSee(Visitor,$Self,Exhibit)). }

**Figure 7.7.** Physical space rules of both Visitor and Room Agents

organisational layer. Note that the visitor agents V1 and V2 are the only agents that appear in both structures.

For the visitor agents to receive preferences, they forward each \texttt{canSee/2}

INTEREST GROUP AGENT
preferences: {
    START => go().
    go() => NEXT prefer($room1,$exhibit1,$exhibit3).
    go() => NEXT prefer($room1,$exhibit3,$exhibit2).
    go() => NEXT prefer($room1,$exhibit1,$exhibit2).
    go() => NEXT prefer($room2,$exhibit6,$exhibit5).
    go() => NEXT prefer($room2,$exhibit6,$exhibit4).
    go() => NEXT prefer($room2,$exhibit5,$exhibit4).
    prefer(X,Y,Z) => NEXT prefer(X,Y,Z).}

request: {
    receive(canSee(Visitor,Room,Exhibit))
    => NEXT canSee(Visitor,Room,Exhibit).
    canSee(Visitor,Room,Exhibit), prefer(Room, Exhibit1, Exhibit2)
    => NEXT send(Visitor,prefer(Room, Exhibit1,Exhibit2)). }

**Figure 7.8.** Organisational space rules of Interest Group Agent
message to their context. The interest groups then reply by sending a preference relation over the exhibits, or alternatively exhibits that should be excluded (Fig. 7.8). Exclusion is accomplished simply by sending a discard/1 message. The agent receiving an exclusion message will go from not having seen the exhibit to seen, without ever making true the predicate goLooking/1 that represents the agent’s action of looking at the exhibit (see Fig. 7.9). Note the message prefer/3 is followed by making the internal predicate prefer/2 true in the next moment of time. The visitor agent will try to honour the eventuality goLooking/1 in the order given by the (set of) preferences. Note that several interest groups can send their preference relations, the visitor agent will internally try to make the order as consistent as possible.

Eventualities are generally attempted in the order they were created. The primitive prefer/2 can change that order. Given a set of prefer/2 predicates, the agent tries to satisfy the constraints they represent. Also note that the order of eventualities is the same across moments in time, so it generally is sufficient to call prefer/2 only once.

In our scenario, the rules that accomplish this modification of eventuality order can be found in Figures 7.8 and 7.9. The visitor invokes interest groups by forwarding to them any information about exhibits it can see. Interest groups simply answer by sending preference relations over the visible exhibits. (Note that executing send(V, prefer($room1, X, Y)) will send all preference predicates that match $room1). The rules of the visitor look complicated because the visitor, after learning which exhibits there are, has to remember those for some time while requesting preferences from the interest groups. During that wait, we must ensure that the eventualities are not honoured.

Also note that while (in this simplified setting) the agent takes only one moment in time to actually look at the exhibits, it still needs to “remember” which
exhibits it should exclude. The exclude rules ensure that discarded predicates are remembered as long as is necessary.

**VISITOR AGENT**

preference: {
  receive(prefer(Room,Exhibit1,Exhibit2))
  => NEXT prefer(Exhibit1,Exhibit2).
  canSee($Self,Room,Exhibit) => SOMETIME goLooking(Exhibit).
  canSee($Self,Room,Exhibit) => NEXT not(goLooking(Exhibit)).
  send(context,canSee($Self,Room,Exhibit))
  => NEXT wait(2000,waitForPref(Room)).
  canSee($Self,Room,Exhibit) => NEXT not(goLooking(Exhibit)).
  send(context,canSee($Self,Room,Exhibit))
  => NEXT not(goLooking(Exhibit)).
  waitForPref(Room) => NEXT startLooking(Room).
  send(context,canSee($Self,Room,Exhibit))
  => NEXT not(goLooking(Exhibit)).
  not(goLooking(Exhibit)), not(startLooking(Room))
  => NEXT not(goLooking(Exhibit)).
  goLooking(Exhibit), not(discard(Exhibit))
  => NEXT lookAt(Exhibit).
  lookAt(Exhibit) => NEXT seen(Exhibit).
  goLooking(Exhibit), discard(Exhibit) => NEXT seen(Exhibit).}

exclude: {
  receive(discard(X)) => NEXT discard(X).
  discard(X), not(seen(X)) => NEXT discard(X).}

exhibits: {
  receive(canSee($Self,Room,Exhibit))
  => NEXT send(context,canSee($Self,Room,Exhibit)).}

---

**Figure 7.9.** Organisational space rules of Visitor Agent

### 7.2.2 Discussion

In above example, while being rather simple, still highlights several aspects of both elements, the structure of the agent space and the use of temporal logic.

For one, the graph-like structure of the agent space can be exploited to contain information about the system. In the above example, the room agents do not know which exhibits they contain until they send a request. The agent space can be very dynamic, and agents do not need to have complicated mechanisms to ensure their internal representation of the world is accurate.
Secondly, not only can we use the structure in such a way, but we can represent different aspects of a system within the graph, design them independently, and combine them at run time. Given the rules in Figures 7.8 and 7.9, we can easily add more rooms, exhibits, visitors, and interest groups, without having to re-write or re-design the system.

The use of logic allows us to extend the system without having to change anything — for example, we can define just the physical structure, which would make the agent to randomly visit the different exhibits. By adding a simple rule that sends requests for preferences when a \texttt{canSee} predicate is received, this can be adapted.

\textbf{Dynamic Aspects: Mobile Agents}

In order to keep the example simple, we assume that some tracking agent tracks the visitors (in the real world) and sends $\text{moveTo(Visitor,Room)}$ messages to the visitor agents. While we omit this in the current example, visitors can easily also be members of this tracking agent.

Figure 7.10 shows the flow of messages that occur when an agent moves from

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figures/7.10.png}
\caption{Messages sent when moving to a room}
\end{figure}
one room to another\(^2\). Upon receiving `moveTo/2`, agent V1 sends appropriate `addToContent` and `removeFromContent` messages to the new and old room, respectively. The `addToContent` message exchange (see Figure 5.3) ends with `addedToContent`, which, in turn, gives rise to the above described interchange between room and visitor which results in the visitor learning about the exhibits available in the (new) room. Note the second rule for the visitor in Figure 7.5, which basically ensures that the agent “forgets” exhibits it might not have looked at in the room it just left. Also note that we can add and remove exhibits “on the fly”, because the room agents always check what exhibits are available.

The movement of agents is independent of other agents being in the rooms, because even though messages are often broadcast to `Content` or `Context`, they generally contain the name of the receiving agent, so that only that agent’s rule will fire. While we could have made sure that messages are only sent to particular agents, this would not have allowed us to (at a later stage) take advantage of the ability to overhear messages within certain groups.

**Dynamic Aspects: Modifying Interests**

We obtain more interesting interactions when looking at the organisational structure. Visitor agents can “subscribe” to interest groups, which in our example determines the order in which the exhibits should be shown. In more complex settings, interest groups also determine or influence the type of information that the visitor will receive during her stay in the museum.

While our example is simple, we can already distinguish several situations. In the first, the visitor is not subscribed to any group; next, a visitor can subscribe to one or more interest groups that give (possibly conflicting) preference relations; she can subscribe to interest groups that suggest avoiding certain exhibits; and,

\(^2\)We omit “send” and abbreviate some of the messages for readability. Also, note that “movement” refers to virtual, rather than actual movement.
finally, a combination of the previous two.

The interaction between visitor agents and interest groups works as follows (see Figure 7.11). After having received different exhibits that are available (\texttt{canSee/3}), the visitor re-broadcasts them to its context, and waits a specified time for answers (\texttt{canSee/3 => NEXT wait/2}). The rule in Figure 7.9,

\begin{verbatim}
not( goLooking(Exhibit)), not(startLooking(Room)) => NEXT
not(goLooking(Exhibit))
\end{verbatim}

ensures that the eventualities \texttt{goLooking/1} will not be made true until the predicate \texttt{startLooking/0} is true.

If the visitor agent is not subscribed to any interest groups (that is to say, there are no interest groups in the visitor agent’s context), it will still wait for preferences. If none are sent, it will just work through its eventualities in a random order. However, if the agent received one or more \texttt{prefer/3} messages, it re-orders the outstanding eventualities using the internal predicate \texttt{prefer/2}. In the case of receiving \texttt{discard/1} messages, our visitor just disregards the exhibit, even if it has a high preference. It should be clear though that we could easily add some rules to, for example, give discarded exhibits a very low preference.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{message_exchange.png}
\caption{Message exchange with interest groups}
\end{figure}
7.3 Travel Agent Example

The ubiquitous travel agent example shows how agents can use different grouping structures as described in Section 5.4.3.

7.3.1 Setting

The setting is as follows. Our agent (A) has the task to book a cheap flight from x to y. A does not know how to do that, so it tries to enlist the help of cheap-flights-agents \((F_1, \ldots, F_n)\), possibly creating, or using, a group agent \(G\) to achieve its goal.

Grouping using Content

\(A\) is the personal assistant of a manager in a large international corporation \(\alpha\). As \(A\) has to deal with many flights, and can negotiate special rates with the flight agents, it invites all \(F_i\) offering flights into its Content. That way, \(A\) can send requests to all \(F_i\), and negotiate with certain flight agents for cheaper flights without the others knowing. \(A\) can also withhold the offers that \(F_i\) makes from all others. Effectively, \(A\) has incorporated the ability to book flights.

Group in Agent

\(A\) is the personal assistant of a manager in an international corporation \(\beta\), but \(\beta\) does not have the buying power to negotiate special deals, so \(A\) creates a dedicated flight group agent \(G\) in its Content, invites all agents \(F_i\) that offer flights to \(G\), and from that point on treats \(G\) as its single flight agent — whenever \(A\) needs to book a flight, it sends a request to \(G\), which simply selects the cheapest offer that its content agents make. \(G\) is “private” to \(A\).
Agent in Group

A is the personal assistant of a manager in a small firm \( \gamma \). As money is precious, and A is able to communicate with the scheduling agents of the employees, it creates a dedicated flight group agent \( G \), but in its Context, i.e. it joins the group itself. That way, it can easily request flights for different dates (or choose different \( F \) agents for different legs of flights), as it receives all messages that occur within \( G \). If it receives a good offer, it can communicate with another group it is member of, such as the scheduling agents group, to see whether the offer fits in the overall schedule.

A not only functions as a member of a group, it can also filter messages from one group to the other, effectively linking groups while keeping the amount of messages that agents receive to a minimum.

Group outside Agent

A is the personal assistant of a local firm, whose employees rarely travel. Whenever A needs to make a booking, it sends the request to a flight agent group \( G \) which it discovered when it was looking for flight agents for the first time. As it rarely needs this ability, it only keeps a reference to \( G \), rather than joining or incorporating it.

7.3.2 Discussion

The example highlights a difficult issue in multi-agent computing and organisational development. It might be clear that even when concentrating on one particular problem (such as a travel agent), different environmental conditions require different ways of tackling the problem. By providing users the choice of deploying different group structures, the problem can be dealt with in a way that is adequate for the given environment. It should be noted here though that while
the above example does provide rather clear distinctions between different situations, it might be much less clear in “the real world” which solution to choose. This not only applies to system designers but specifically to software agents who should be able to choose appropriate group structures dynamically.
Chapter 8

Related Work

In this Chapter, we will endeavour to situate our work within the wider community. It should be noted that we are not aiming at giving a comprehensive comparison with the substantial body of work produced in the fields of Logic, Artificial Intelligence, and Computer Science. Instead, we choose to highlight some of the areas more closely related to our work. In particular, we will highlight some group based approaches in Sections 8.1 and 8.2, and discuss the Java-based multi-agent system JACK in Section 8.3. Finally, we will provide a detailed comparison of our work to a contemporary mobile calculus in Section 8.4.

Most of the research described here was mentioned earlier in Section 4.5, but now we will highlight differences and similarities with our approach.

8.1 Holonic Agents

While considerable work has gone into the formalisation and analysis of teamwork and team building, there is a much smaller body of work related to structuring the agent space within a multi-agent system. One system that bears a close relationship with our structured agent space is the work on holonic manufacturing and holonic multi-agent systems [Dee94, JJM03, FSS03, Sch03]. Holonic agents
are defined recursively. Each holon is defined in terms of a set of other holons, which, at the lowest level, consist of basic elements. Holonic manufacturing was developed to model modern production techniques. While traditional factories where built with the intention of building many similar products as efficiently as possible, today’s requirements are changing in that more individualised products are required [JJM03]. By representing elements of the production line in terms of holons that combine to achieve more and more complex elements, production can be streamlined (see e.g. [LCR94]). In a manufacturing environment, holons are typically structured in a highly hierarchical environment, and even though different holons might be constructed in order to fulfill production needs, they all exist on a defined layer, rather than moving through the space as needed. Based on work on holonic manufacturing holonic agents were developed [FSS03, Sch03]. Similar to our work [HFG03a], holonic agents can form different types of organisations [Sch03], ranging from loosely coupled agents (independent autonomous agents) to very rigid and hierarchical systems (corporations). However, holons are purely theoretical constructs that are constructed by agents communicating and agreeing to a structure, rather than representing the actual structure within the multi-agent system [Sch03].

8.2 Group Based Approaches

We will in this section discuss team (and group) based approaches to multi-agent systems, such as MadKit, which implements the AALAADIN model, and an organisation-oriented approach [Tid99], which is based on joint intention theory.
8.2.1 AALAADIN

Ferber and Gutknecht [FG98, GF98, Fer99, FGM03] have developed a modelling technique for multi-agent systems called AALAADIN [FG98] which is based around groups. They also provide an implementation (MadKit) [GF00] which implements these group structures in Java. Agents, playing roles, are partitioned using groups. Groups themselves are described using a group description which consists of roles that constitute that group. Agents cannot communicate outside groups, but may be playing (different) roles in (different) groups [FGM03]. However, the system has drawbacks that stem from the fact that groups are rigid, and are not themselves viewed as agents. In [FGM03], for example, Ferber and Gutknecht give an example of an agent joining another group. In order to do so, the group needs an “entrance group” which has no restrictions as to who may join, and which needs to be populated by a “gatekeeper” agent which then allows or disallows the agent to join the actual group. In our framework, the group itself, as it is an agent, can take over negotiations concerning group membership. Furthermore, the language we devised to allow communication across group borders takes away the pressing need for agents to create new groups to communicate (though, as discussed, if they do send many messages or are far apart, it does make sense to create a dedicated group for that). Finally, Ferber’s agent space appears to be basically a projection of a structure onto a one-dimensional group-layer, rather than a direct representation of the structure of the agent space.

8.2.2 Organisation Oriented Systems

In [Tid99], Tidhar proposes a organisation-oriented approach to designing multi-agent systems, which as been extended in [TS03]. He proposes to design multi-agent systems with organisations as the basic entities. The underlying semantics is based on joint intention theory [LCN90], which we have described earlier. In
such a model, the system designer controls the relationships between components in the system. Organisations are made up of sub-organisations, though this concept exists solely as the beliefs that the sub-organisations have with respect to each other, rather than a “physical” existence within the organisation (as opposed to our own approach). However, the organisation itself has its own beliefs, as it is a basic entity, and not just a result of joint beliefs or intentions of the sub-organisations. Where we base our system on the MetateM approach, Tidhar extends the BDI theory of Rao and Georgeff and uses an extension of CTL, augmented with modalities for beliefs, goals, intentions for single agents, and joint intentions as well as team operators to model the multi-agent case. This leads to a powerful logic in which organisational structures and their relationships can be expressed. While an implementation based on dMARS [Kin93] is provided, the link between specification of the system and the actual implementation is not clear.

Tidhar recognises that the addition of organisational structures to multi-agent systems is a valuable asset, in terms of systems design, as well as implementation of a multi-agent system. He also sees the need for a logical basis. However, where we believe that a less complex approach, using an executable logic, and the identification of group and agent is prudent; Tidhar’s work is based on the notion that powerful logics lead to a better understanding of multi-agent systems.

8.2.3 Practical Applications

As an example of the application of structured agent-systems we want to discuss the work of Zhang and Lesser[ZL04]. They experimented with dynamic hierarchical structures for content sharing systems (p2p software). We mention this work as it deployed a system similar to our Content/Context approach, and allowed agents to move within the structure.
Similar to popular p2p-systems, each agent contains a local database. Users can query “their” agent for some content, which it will first try to find in its internal database, and then in other agents. Agents are structured in a hierarchical structure. However, rather than just allowing for links up and down, Zhang uses three sets of communication links, one for communicating upwards, one for communicating downwards, and a third one for communicating with peers that are on the same level.

Agents form groups based on similar content. Each topic has a top-level agent that contains the sub-agents, and decides on whether to include new agents in its content. Agents can be members of several other agents as well.

In order to maximise the advantage that structuring provides, agents only allow a certain (small) number of agents to be in their “content”. If the number is exceeded, agents in the content re-group to add an additional layer, i.e. some of the agents move into the content of one of the neighbouring agents, thereby extending the structure.

Zhang’s et al. agents very closely resemble our structure. While they use a third set of communication links to communicate with peers, we do so by way of routing messages through the context. However, we could use the Known set to model neighbouring links, and would arrive at an identical structure.

### 8.3 Agent Frameworks

JACK [BRHL99] is arguably the best known (commercial) agent architecture available, so we will use it as representative of established agent frameworks. Furthermore, JACK has some similarities to our approach.

Currently at version 4.1, the framework provides a framework for Java-based agents. JACK supports a BDI-like structure, providing support for databases that contain beliefs, goals, and plans. Through the use of extensions to the Java
programming language, agents can quickly be defined in terms of their capabilities, the events they listen to, their beliefs, plans, and so on. While not providing a formal semantics for the BDI part of the language, JACK defines a Prolog-like interface to query the belief and plan databases. Recently, JACK agents have been extended with a team model, called SimpleTeam [HRHL01]. SimpleTeam adds the concepts of role and teams, as well as extending plans and capabilities to include teams as well as agents. In fact, the current manual [JaT04] describes teams (which seem to be an extension of SimpleTeams) as “individual reasoning entities”, thereby elevating the team concept to something that comes very close to our identification of agents and groups. Also, teams “live” independently of their members. For example, they can reason about which agents or sub-teams to include or how to distribute roles that need to be filled within the team.

JACK’s team extension allows for quite intricate team or group structures, similar to our framework. However, while supporting beliefs and goals, JACK does not provide a clear semantics of the behaviour of the BDI part of the implementation. Also, while essentially defining teams as an extension of agents, they stop short of actually identifying the two notions. Although teams can access and use “normal” belief- and plan bases, the system seems as of yet unable to allow BDI behaviour for teams [JaT04].

8.4 Milner’s Bigraphs

We now look at other theoretical approaches to groups, agents, mobility, and so on. As a representative, we will introduce the concept of bigraphs [Mil04], and show that our structured agent space has a connection with mobile calculi. While there are many calculi available that model (mobile, communicating) processes, we chose to use bigraphs as it subsumes other calculi, such as the ambient calculus [CG00], or the π calculus [MPW92].
Bigraphical reactive systems are a graphical model of computation in which both locality and connectivity are represented. While there is a textual way of specifying bigraphs, its basic form is graphical.

Milner shows, in [Mil04], that bigraphs can represent dynamic theories ranging from π-calculus and mobile ambients to Petri nets. While he focuses on rigid mathematical proofs to show equivalences, we will try to give the reader an intuition of how bigraphs can be represented using our structured agent space and vice versa.

Technically, bigraphs are a combination of two distinct graphs that share nodes. Nodes can be nested, creating a place graph that has depth. Nodes can have ports, which in turn can be connected by links that essentially partition the ports, creating a link graph. In the following, we will introduce the two elements independently, show how the structure can be emulated using our structured agent space, and finally show that full bigraphs also can be represented.

8.4.1 Place Graphs

Place graphs consist of regions which contain nodes, which in turn can contain other nodes. Regions and nodes can also contain “holes”, which represent slots in which one can add other bigraphs. The number of holes in a place graph determines its inner width, whereas the number of regions is called the outer width of a place graph. We call the holes sites, and the regions roots. In the example shown in Figure 8.1, we can find one root, and no holes.

Nodes can also be assigned a control, which governs the behaviour of the nodes. Controls have an arity, i.e. a number of ports which can be connected by a link graph. Conceptually, nodes can represent a multitude of entities, from agents to cryptographic keys.

Formally, a place graph is defined as follows, given a signature $\mathcal{K}$ containing
controls, and using $S \uplus T$ to denote a union of sets known or assumed to be disjoint.

**Definition 32 (Place Graph)** A place graph $G = (V, ctrl, prnt) : m \rightarrow n$ has an inner width $m$ and an outer width $n$, both finite ordinals; a finite set $V$ of nodes with a control map $ctrl : V \rightarrow K$; and a parent map $prnt : m \uplus V \rightarrow V \uplus n$. The parent map is acyclic, i.e. $prnt^k(v) \neq v$ for all $k > 0$ and $v \in V$.

This definition yields a forest of $n$ unordered trees. Place graphs can be combined by connecting the roots of one with the sites of another place graph.

### 8.4.2 Link Graphs

Link graphs connect ports and places. They represent connectivity, as opposed to locality which is represented the place graphs.

Link graphs consist of edges that connect ports. A link can connect an arbitrary number of ports. If an outer name is connected to the link, the link is considered to be open, i.e. it can merge with other links when combining bigraphs. Figure 8.2 shows a link graph with one open link called $y$.

Formally, a link graph is defined as follows, again using $S \uplus T$ to denote the union of two disjoint sets, and $ar$ is a function returning the arity of a control.
Definition 33 (Link Graph) A link graph \( G = (V, E, \text{ctrl}, \text{link}) : X \rightarrow Y \) has finite sets \( X \) of inner names, \( Y \) of outer names, \( V \) of nodes, and \( E \) of edges. It also has a function \( \text{ctrl} : V \rightarrow \mathcal{K} \) called the control map, and a function \( \text{link} : X \uplus P \rightarrow E \uplus Y \) called the link map, where the disjoint sum \( P \overset{\text{def}}{=} \sum_{v \in V} \text{ar}(\text{ctrl}(v)) \) is the set of ports of \( G \).

8.4.3 Bigraphs

While the two graphs are orthogonal to each other, they do share a common set of nodes and their associated controls. Therefore, it is straightforward to combine the two graphs in order to get a bigraph. Figure 8.3 shows the combined place and link graphs depicted in Figures 8.1 and 8.2 respectively. Note their common elements are the nodes \( u_0 \) to \( u_5 \) which allows us to combine the two graphs.

Definition 34 (Bigraph) A bigraph over signature \( \mathcal{K} \) takes the form \( G = (V, E, \text{ctrl}, G^P, G^L) : I \rightarrow J \), where the interfaces \( I = \langle m, X \rangle \) and \( J = \langle n, Y \rangle \) are its inner and outer interfaces. Its first two components \( V \) and \( E \) are finite sets of nodes and edges, respectively. The third component \( \text{ctrl} : V \rightarrow \mathcal{K} \), called control map, assigns a control to each node. The remaining two graphs, place and link graph, are defined in Definitions 32 and 33, respectively.
While in Milner’s description [Mil04], controls only relate ports and nodes, he allows for the possibility of extending them by various means, such as sign or type. The more elaborate the control is, the closer it comes to our definition of agent, where elaborate rules govern how agents interact.

8.4.4 From Agents to Bigraphs

Bigraphs consist of a combination of (multiple) trees (place graphs) and graphs (link graphs), each with connections to the outside. The two graphs are to be interpreted as the location of controls and their connections, respectively. Our goal in this section is to define a mapping from our agent structure onto bigraphs. While it should be clear from the description that such a mapping is possible just in terms of link graphs, this mapping would be rather uninteresting.

Place graphs however, are not able to directly encode agents being members
of more than one group. When we restrict our system though to allow only tree-structures, i.e. each agent can be a member of only one group, we have a direct link between bigraphs and our structured agent system as follows.

Agents are represented by controls that have one site which represents an open slot in the agent’s *Content*. The system’s third communication set is represented by the link graph, which allows us to implement communications. Using this simple mapping, we can now create reaction rules that mimic the behaviour of agents within the agent system. Figure 8.4 shows a possible representation for the `moveInto` behaviour.

![Reaction Rule for Agents](image)

**Figure 8.4.** A reaction rule for agents

For space considerations we have abbreviated `moveInto` to “in”. Also, we assumed a link connection between the two agents (named `w`). Note the grey boxes which represent the different sites, or holes into which other agents can slot. In the reaction rule, the receiving agent creates a new “hole” in order to be able to receive other agents as well.

Note that the figure essentially is also a representation of a reaction in the *ambient calculus* [CG00]. However, there, agents (or ambients) are not able to communicate outside their nesting. If we further restrict our agent system to
not make use of the third set, we arrive at a correspondence between our agent based system and ambients. The above reaction rule can be expressed in ambient calculus as $P[\text{in}]Q \rightarrow Q[P[]]$.

Bigraphs, and in particular place graphs, can consist of more than one region, and it seems compelling to allow for agents (being represented as bigraphs) to split into multiple regions in order to be able to join more than one group, but while this works for a static bigraph, it is by no means clear how such a split agent could engage in rules such as that shown in Figure 8.4, as the number of regions (resp. sites) would be unknown. More research into connections between our structured agent system and bigraphs is necessary to see how this can be overcome.

### 8.4.5 From Bigraphs to Agents

In this section, we will show how we can map important aspects of Milner’s bigraphs onto the nesting of agents within our approach. First of all, we assume the bigraph itself to be an agent, in order to facilitate communication. In order to map a bigraph onto a structured space, we have to show how we can map the place and link graphs.

Milner introduces several distinct entities, which we will have to map onto our agent space. The two most obvious ones are nodes and links. Both of these are associated with ports. Nodes themselves can be nested, or can contain holes, where other nodes can slot in, provided the outer names and inner names of the node to be slotted in and the node itself respectively match. Ports are part of both links and nodes. Each port can only connect to one link. Ports can also have names, in the case of inner or outer links.

Our structured agent space provides agents that have three distinct sets of connections, namely \textit{Content, Context,} and \textit{Known}. Agents also have behaviour
that governs how agents communicate and move within the agent space. Both
the structure and the behavioural element are expressive enough to at least map a
substantial subset of Milner’s bigraphs. In the latter case, we define a flat agent
space, in which all agents broadcast all messages, and the rules within agents
simulate place and link graphs. We will not pursue this any further, as there
are undoubtedly programming languages that are better suited for that task. On
the other side, we can try and create bigraphical structures using as few rules as
possible, essentially mapping each entity onto its own agent (including ports and
names), and creating a multi-layered structure that represents the connections
that bigraphs can contain. This approach, while more suited to showing the
strength of our approach, also has practical drawbacks. For one, we will have to
have some sort of behaviour for agents in order for them to distinguish between
the different roles they are playing. More importantly, perhaps, the resulting
structure would be rather complex, with communication always going through
several layers of agents, and becoming too “artificial” and unwieldy.

Instead, we opt for a hybrid solution, harnessing the power of the structure,
yet allowing (simple) structural elements to be represented by behavioural rules
within agents. The result shows that bigraphs can naturally be mapped onto our
structured agent space.

Mapping the Place Graph

The easy part of the mapping is the place graph, as this corresponds very closely to
our Content/Context relation. The requirement of acyclicity is met by requiring
agents to have a rule allowing at most one parent, or one agent in the Context set.
Rather than using agents to represent holes, we assume that nodes that contain
holes have a rule defining both the hole and the ports connected with that hole.
While, intuitively, we might want to associate a separate agent to represent holes,
this would add to the complexity of merging different bigraphs without adding much clarity. Note that holes disappear when bigraphs merge, therefore the hole agent’s task can almost trivially be taken over by the “owning” hole. Similarly, ports, when viewed from the place graph, are very passive entities, therefore also not warranting representation by agents. Figure 8.5 shows an “agentified” representation of the place graph shown in Figure 8.1.

![Figure 8.5. Agents representing place graph](image)

**Mapping the Link Graph**

Link graphs are somewhat more difficult to represent. For one, links essentially are sets of ports, rather than sets of nodes, which would fit into our mapping of place graphs. Furthermore, those ports can be connected across different nestings of nodes.

Defining links in terms of sets of ports suggests elevating ports to agents, and defining links as agents containing ports, which in turn are contained by not only
the link but the node agent they belong to. However, as ports have no behaviour, and merely facilitate the connection between nodes, we think that the resulting representation would be too “cluttered” with agents that do little more than serve as containers for other agents. Therefore, we choose to represent links as agents that contain the nodes that are connected to the link. Links are meant to represent connectivity, and this is neatly achieved by combining agents that share a link in a link agent. Communication can be carried out by broadcasting messages through the link agent.

We therefore map links to agents, and nodes that are connected to the link reside in the link’s Content. Furthermore, the link will have rules defining whether or not it is open, i.e. whether it is able to merge with other links in the event of bigraphs being combined.

Figure 8.6 again shows an “agentified” version of the link graph as depicted in Figure 8.2. Note that we omit here the open link $y$, as this information is hidden in the rule set of agent link$_1$. Also note that we assume node $u_0$ to be member of a link agent containing just itself.

![Figure 8.6. Agents representing link graph](image-url)
Alternative Mapping  With the mapping described above, node agents will have two distinct types of agents in their context, namely node agents and link agents. This multiple use of structure is part of what gives the structured agent space its power. However, we could, in order to emphasise the orthogonality of link and place graph, make use of the set Known that agents have. Originally, the Known set, separate from Content or Context, was implemented to facilitate communication in complex structures, as it allows agents to “jump the queue” and send messages directly rather than via (potentially many) other content/context links. Conceptually, however, the Known set is no different to Content or Context. An alternate representation, therefore, could connect nodes and links not via content/context relation but via direct relations.

Mapping the Bigraph

As the theory of bigraphs defines the link and place graphs as distinct entities that share a common set of nodes, the mapping of a complete bigraph based on mappings of its constituent graphs should be straightforward. Certainly given the alternate mapping of link graphs, we can simply identify node agents, and the two graphs will still be represented. But even if we choose not to use the Known set but instead map link graphs to Content/Context relations, we have to do little but identify node agents and combine respective Contents and Contexts. Again, the reader can find a picture of the resulting agent system in Figure 8.7. Note, again, that certain information that is encoded within the agents, such as names, is hidden. Also, we choose to omit a outer agent representing the whole bigraph.

8.4.6 Dynamic Bigraphs

In the last subsection we have shown that bigraphs can be mapped onto structured agent space without too much trouble. However, while bigraphs themselves might
be interesting mathematical constructs, their versatility and power come to light only when we allow for bigraphs to interact.

Milner characterises bigraphs as “arrows” of a partial strict symmetric monoidal category whose objects are simple forms of interfaces. In this work, we will not go into the intricacies of defining categories that represent the calculus. Instead, we focus on how the transition rules that transform bigraphs can actually be implemented. Ultimately, we would aim to provide a language where one can define bigraphs and transitions, and can compute interactions using structured agent space.

As before, we will examine composing place graphs and link graphs independently as, due to the orthogonality of the graphs, the combination is rather trivial.

**Composing Place Graphs**

Place graphs have roots and sites. Combining place graphs simply means connecting roots and sites. Following Milner’s definition (Def. (32) on page 182), roots and sites are represented by ordinals, and matching roots connect with respective sites. In the case of agents, we would probably choose to name roots, rather than
describing them using numbers. Furthermore, sites are not represented as such, but are defined within nodes themselves.

To combine two agents representing place graphs, we first assume that they are both members of a system agent. While the precise structure is not important, we must ensure that the two place graph agents can communicate via some path. For the rest of this section we just assume that they can send and receive messages to and from each other. We also ignore possible protocols and internal deliberations of graph agents, and assume that agents have the intention to combine their respective graphs.

There are many possible protocols to achieve the merger of two place graphs. One such protocol might look as follows:

If graph $G$ wants to connect to $H$ (meaning that $H$ will be providing the root of the resulting graph), it sends a message $\text{merge}(G,H)$ to $H$. $H$ responds by broadcasting a message $\text{nameSite}(G)$ to $G$, asking for the names (or ordinals) of the roots that $G$ consists of. Furthermore, it sends a message $\text{hasSite}()$ to all nodes within its content. Nodes that receive $\text{hasSite}()$ either pass it on to their Content (if they do not have a site), or they respond by sending $\text{site}(\text{Node}, \text{Name})$ back to their Context. Again, nodes receiving that message just pass it on to their Context, until $H$ receives the messages. Now, when $G$ answers with $\text{hasSite}(G, \text{Node}, \text{Name})$, $H$ can check whether all the sites of $H$ have appropriate roots in $G$. If so, $H$ sends a $\text{connectTo}(G, \text{RootNode}, \text{SiteNode})$ to $G$, for each root that needs to be connected to a site. Now, the nodes themselves can communicate, and simply add the root node to the content of the site node (and vice versa). $G$ itself is empty, and its execution can stop.
8.4 Milner’s Bigraphs

Composing Link Graphs

Link graphs are composed by matching the inner names of one link graph with the outer ones of the graph that will be slotted in, and merging matching links. Note that there is no requirement for all links to connect in some way. Links of one graph whose outer name is equal to the inner name of a link of the other graph are merged, resulting in a graph that contains all the ports of both links, as well as the inner names of the link that is slotted in, and the outer names of the link that serves as host.

Similar to place graphs, we assume that the merging link graphs can communicate via some route. We call the link graph that is slotted in $A$, and the host graph $B$. However, where merging place graphs meant connecting appropriate agents through their $Content$ and $Context$ relations, link graphs are more complicated. Recall that we represent links using agents. When combining link graphs, we connect links that have the same inner and outer names. Therefore, if the combining link agents have matching inner and outer names, they need to merge, i.e. combine their $Content$, $Context$, and names (other than the names used to combine, which will disappear).

8.4.7 Conclusions

In this section, we have shown that there is a clear connection between our work and the more theoretical research into mobile calculi. Bigraphs can be mapped onto our structured agent space, thereby providing a modelling tool for the language. While the other direction is not as straightforward, hierarchical structures can be mapped onto bigraphs, as well as agent movement. It is our hope that, with more research, we will be able to complete the mapping, thereby providing a sound theoretical basis and language for the structured agent space.
Chapter 9

Evaluation

We conclude this thesis with an evaluation of the work carried out, and a discussion of directions for future research.

9.1 Conclusions

In this work, we have presented a multi-agent programming language that stands on two pillars. Firstly, agents are programmed using a directly executable temporal logic that has been enhanced with bounded belief, abilities, and deliberation. Secondly, the agents “live” within a structured environment, and in fact are the structure, allowing the programmer to use sophisticated structures to organise systems.

In the course of this work, we have created an implementation of the multi-agent framework, using extended METATEM to program individual agents. We have provided numerous examples that show the merits of approaching the problem of programming complex and dynamic systems using a structured agent space. Furthermore, we have extended the language to allow programming deliberation within the language, rather than doing so on a meta-level.

While we could show that METATEM, even with its extension of bounded
belief, is indeed a feasible way of programming agents, it must be said at this point that, depending on the program, the computational resources needed to run agents are still rather large. Using some programming tricks that we have implemented, such as the use of variables, mitigates this somewhat, but at the moment, the provided implementation would not be able to cope with a serious implementation, not only because it does not allow agents to be distributed, but due to the search space that larger computations create. Having said that, it should be clear that the current implementation was developed with the aim of providing a test bed for the presented theories, and that future iterations should be able to handle complex agent systems more efficiently.

We have provided an operational semantics for single METATEM agents with belief, as well as for multiple agents executing within a structured space. We have shown a close relationship between our structured agent space and the bigraph calculus, allowing to use them as a formal foundation of part of our system on the one hand, and to implement bigraphs using agents on the other.

Within the mentioned boundaries, we believe to have shown that it is feasible to program agents using temporal logic. Thanks to the possibility of using code blocks which allows the programmer to use default behaviours while still defining an agent’s individual behaviour, systems with many agents that are based on the same set of rules can be constructed quickly and easily.

Furthermore, we made a case for using structured agent spaces to implement complex systems. While it is a commonplace to note that corresponding behaviour can also be programmed using systems that do not specifically allow for structured spaces, its elevation to an integral element of the agent system allowed us to more easily map structures used in modelling systems onto an actual implementation of such a model. We have shown how the structure can be exploited to create complex systems that are (a) relatively easy to specify, due to
the possibility of designing different layers independently of each other, (b) dy-
namic, and therefore suitable for systems where many different agents interact in
unforeseeable ways, and (c) potentially verifiable, due to the logical basis of the
system. The key behaviours of individual agents are provided through varieties
of executable temporal logic, while the over-arching group structure allows us to
represent a range of physical and virtual organisations. This approach provides a
powerful, flexible, yet logic-based, route to the design, modelling and development
of software for ubiquitous computing applications.

9.2 Future Work

This work is only a first step in the exploration of both the actual usage of ex-
tended MetateM as a language to implement agent systems, and the use of
structured space to represent problems. Many avenues of future research can be
identified at this point. On the implementation side, we should add networking
capability, as well as a graphical interface to facilitate the programming and mo-
itoring of agent systems. We have in this work not focussed on the efficiency of
the implementation, and no quantitative experiments with respect to speed, num-
ber of messages used, memory requirements etc. have been undertaken, things
clearly necessary before we could “sell” the implementation as a viable modelling
tool or multi-agent-environment.

The theory of a structured agent space was, in this work, based on a basically
hierarchical framework, where agents are connected “upwards”, via their Context,
and “downwards”, via their Content. However, we have seen that the agent space
can be used to map multiple different structures simultaneously. It therefore
seems only logical to expand the notion of Content and Context such that agents
can have multiple “clusters” of agents, leading to a multi-dimensional structure of
connected agents. Whether this really leads to a more clearly structured system
or to more chaos is something that needs to be examined.

While METATEM has been extensively studied, some elements still need attention. This includes a (temporal) semantics of multiple (potentially backtracking) agents running concurrently (something which has been touched in [Fis96b]), as well as a further investigation into a temporally bounded extension of bounded beliefs (which has also already been suggested in [FG02b]).

While we have provided an operational semantics for the agent space, it is clearly desirable to either develop an extension to the temporal semantics which encompasses the structured space, or to more rigidly define mappings between the structured space and established semantics, such as process algebras. It might also be possible to define a modal logic which interprets the agent structure as a kripke-style model. A mathematical investigation into properties that define the structure might also be of interest.
Bibliography


[BDH+01] J. Broersen, M. Dastani, Z. Huang, J. Hustijn, and L. van der


Appendix A

Documentation

This chapter contains the documentation of structured METATEM. After sketching the general structure, we will give a bottom up description of the programming language. We start with the smallest element, the predicate (Section A.2), expand to rules (Section A.3), and end with outlining the structure of the files needed to get Structured METATEM running. Finally, we present the default constructs that implement many of the movements of agents discussed in Section 5.4.

A.1 System Description

Structured METATEM is a programming language for multiple agents. The current version, while fully functional, misses important aspects of multi agent systems. Most prominently, agents have to “live” within the same Java virtual machine, it is not possible to distribute agents over a network. This curtails the possible usage of Structured METATEM, but as we envision it to (at this stage) be used mainly for rapid prototyping and modelling, this restriction will hopefully not deter the user.

As described in the previous chapters, the novelty of our approach lies in the
structuring of the agent space. Agents live within a graph, rather than a flat environment. They can move within that structure, add and remove vertices (agents) and edges (communication connections).

The structure is represented by each agent having two sets of connections to other agents, their \textit{Content} and their \textit{Context}. (Agents also have a third set called \textit{Known}, which is mainly used for efficiency purposes).

Agents communicate through (sub-sets of their) \textit{Content} and \textit{Context}. The primitive \texttt{send} has two arguments, the first of which describes the set that should receive the message, while the second is the message itself. Structured METATEM does not impose structure on the message, other than that it should be wrapped in a predicate.

During execution, agents try to match true literals as well as received messages (which become literals of the form \texttt{received(message)}) with their rule base. If the antecedent of a rule matches, and no free variables are left in the consequent, the rule will “fire”. Furthermore, in order to keep to stay as closely as possible to the logic, rules that have no free variables and whose antecedent is not inconsistent with the set of literals will fire as well, though with a lower priority (meaning that they will be used only if really necessary). Note that received messages become true as \texttt{receive(message)}, rather than \texttt{message}. In order to make the actual messages true, the agent would need a rule of the form \texttt{receive(X) => NEXT X}.

\section*{A.2 Predicates}

The basic entity of a Structured METATEM program is a predicate. Similar to Prolog, a predicate is a string (beginning with a lower case letter) followed by zero or more terms enclosed in brackets. The regular expression \(~\text{\texttt{([a-z,\_.\[a-z,A-Z,0-9,\_.\-,\.,\_/\]*("\(\text{\texttt{(TERM(,)?\_*\)}}\))\)}}\)
defines predicates. Examples of predicates would be `goNow()`, `add(1,2)`, `wait(X,go_Now("start now"))`.

Terms can be of any of the following subtypes.

- **Integer**
- **String** — Any text (including spaces) enclosed in double quotes (e.g. "text").
- **AgentName** — Any string starting with the dollar sign $. The special name `$Self` refers to the agent that is executing the rule.
- **Variable** — Any string starting with a capital letter.
- **SetExpression** — any of
  - an AgentName;
  - a list of SetExpressions, enclosed in curly brackets "{" "}";
  - `content,context,known` — these are special variables denoting the agent’s content, context, or known set;
  - `SetOperations` on SetExpressions, such as S1 union S2, S1 intersection S2, S1 without S2.

While Structured METATEM tries to recognise SetExpressions, the user is advised to use curly brackets to mark them.

- **Function** — A function is defined the same way predicates are defined.

Variables are denoted by strings that start with an upper case letter. The scope of variables always ranges over a single rule.

Semantically, predicates are nothing but fancy propositions. METATEM originally is a propositional language, and as such does not allow for variables or
predicates. Structured METATEM, while allowing variables and predicates, also is a propositional language at heart. This means the following:

- Variables always range over a single rule.
- Only rules with no free variables can “fire”.
- Only grounded predicates can become true.

Generally, variables can replace anything up to predicates, so one can write rules such as \( X \Rightarrow \text{NEXT} \ print(X) \). and \( \text{receive}(X) \Rightarrow \text{NEXT} \ X \). Note that in the latter case, there needs to be a space between the variable and the final dot (this is a parser restriction).

**Negation**

The special predicate \( \text{not}/1 \) is interpreted as logical not. A not-predicate on the left hand side of a rule will be considered true if the predicate is not known, or if \( \text{not}(p()) \) is true (following the closed world assumption). Similarly, \( \text{not}(p()) \) will not fire if \( p() \) becomes true. One can create rules such as \( \text{not}(\text{received}(\text{message})),\text{keepDoing}() \Rightarrow \text{NEXT} \ \text{keepDoing}() \). in which case the agent will make \( \text{keepDoing}() \) true until it receives \( \text{message} \). Note however in order to use such rules, one is advised to use predicates with variables in them, as rules otherwise might fire “by default” which might or might not be the intention of the programmer\(^1\).

**A.3 Rules**

Structured METATEM is programmed using temporal formulae in SNF form. The following rule types are allowed:

---

\(^1\)This recurring theme is somewhat dual in nature. From a “simple” programming point of view, one wants to be sure when rules fire — namely only if the antecedent is made true by preceding rules. However, from a logical point of view, rules should be able to fire at any (consistent) time.
Generally rules have the form “\texttt{predicates} \Rightarrow \texttt{predicates}.” The “\texttt{.}” terminates a rule. Predicates are are divided either using “\texttt{;}” or “\texttt{,}”. The former represents a logical OR, while the latter is interpreted as a logical AND. A predicate has the form \texttt{name(args)}, where \texttt{args} is a comma delimited list of zero or more predicates, numbers, or strings. In the following, we’ll denote disjunctions of predicates as \texttt{OR-predicates}, and conjunctions with \texttt{AND-predicates}.

\textsc{Start rules} have the form \texttt{START} \Rightarrow \texttt{OR-predicates}. \textsc{Start} rules are executed at the beginning of time. The system initially makes only a subset of the predicates true. To simulate a list of \texttt{AND-predicates}, the user is advised to use multiple \textsc{Start} rules.

\textsc{Next rules} have the form \texttt{AND-predicates} \Rightarrow \texttt{NEXT OR-predicates}. \textsc{Next} rules “fire” if all predicates in the antecedent match with predicates that are true at the given moment in time, in which case a subset of the predicates of the consequent. To simulate a list of \texttt{AND-predicates}, the user is advised to use multiple \textsc{Start} rules.

\textsc{Sometime rules} have the form \texttt{AND-predicates} \Rightarrow \texttt{Sometime predicate}. If all predicates in the antecedent are true, the consequent will be made true as soon as possible. Please note that eventualities are ordered, so the system will try to make older eventualities true first.

\textsc{Belief rules} have the form \texttt{AND-predicates} \Rightarrow \texttt{Belief ((Agent))}? \texttt{OR-predicates}. If the agent is omitted, the agent itself is taken to be the “believer”. In order to assign a truth value to beliefs, the system creates new time lines (for each agent) in which beliefs are assumed to be true and tries to satisfy them. Note that the depth to which the system expands beliefs can be set in the header with the switch \texttt{depth}.
A.3.1 Rule Blocks

Structured METATEM allows the programmer to collect rules that conceptually belong together. Such a conceptual entity is called a rule block. The format of a rule block is name: {rule; . . ;rule;}. If the system reads a rule block whose name already appeared, it will remove all old rules and replace them with the new block. This can for example be used to define default behaviours that reside in a rule file which all the agents load. Particular agents then can easily be customised by re-defining rule blocks.

A.4 File Types

Structured METATEM environments consist of several files. Generally, we will find one environment file (Sec. A.4.1) that describes the initial configuration of the agent space; one or more agent definition files (Sec. A.4.2) which define the behaviour of the different agents; one or more rule files (Sec. A.4.3) which can be loaded by the agents; and external abilities (Sec. A.5), which basically are java classes, possibly with associated rule files.

A.4.1 The Environment File

The environment file defines the initial structure of the agent space (Figure A.1). This is done in two steps. First, the agents that participate are loaded using the agent NAME: FILE; directive. Next, the structure is constructed by assigning each agent’s content and context. The assignment is done using the names given for loading the agent definition files. Note though that NAME is independent of the agent’s name during execution(!) which is defined within the agent definition file.
A.4.2 The Agent Definition File

The agent definition file is the most important file. Here, the behaviour of the agent is defined. The definition file consists of a header, which defines the name of the agent, external abilities and other settings, and a main part, which contains zero or more rule blocks.

The structure of the file is $Header \ (Ruleblocks)^*$, where $Header$ contains meta information about the agent, while the rule blocks contain the actual behaviour rules. Figure A.2 gives an overview of the commands that can be used in the header. The commands in bold are obligatory, while the ones with a (*) can be stated more than once.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>agent name;</td>
<td>defines the name of the agent</td>
</tr>
<tr>
<td>include file;</td>
<td>allows to load rule files</td>
</tr>
<tr>
<td>debug boolean;</td>
<td>toggles debug messages</td>
</tr>
<tr>
<td>statedebug boolean;</td>
<td>toggles execution messages</td>
</tr>
<tr>
<td>depth number;</td>
<td>sets the depth of belief expansion</td>
</tr>
<tr>
<td>ability name:file;</td>
<td>loads external abilities</td>
</tr>
</tbody>
</table>

Figure A.2. List of header commands for agent definition files
agent name  defines the name of the agent as it is known within the system. Agents can refer to each other by their names, though one should note that it is only possible to directly send messages to agents using their names if the agent is known, that is, either within Content, Context, or Known. The system by default moves all agent names it encounters into Known, so that faster communications are possible.

include file  allows to load rules that are defined in another file. This is very useful if we want several (more or less) identical agents, as the common rule base can reside in a rule file. If a rule block appears in an included file as well as within the agent definition file, the latter will take precedence and over-write the included rules.

debug  allows to turn on status information for agents. Note that this can also be done using rules — the internal predicate setdebug/1 ("true","false") (un-)sets the same status messages. The messages printed mainly concern internals of the processing.

statedebug  works similar, but turns on the output of state information — predicates true in the state are printed, as well as the rules that fire. As with debug, statedebug/1 also exists as internal predicate that can be used to selectively turn on or of the output. Note that statedebug output is rather verbose.

depth number  sets the initial depth to which belief contexts are expanded — if no depth is defined, beliefs will be expanded to level 3. There should be an internal predicate that sets the depth dynamically.

ability name:file  allows to include external abilities. "name" describes the predicate that is linked with the object pointed to by "file". Note that the object
A.5 External Abilities

has to make sure that it can deal with the arguments of the ability — there is no restriction as to arity, which means that one object can allow for several predicates (with the same name but different arity). The objects can return predicates as well, which are treated as messages coming from the agent self — this allows the object to spawn new threads or make long computations while the agent can continue its execution.

A.4.3 Rule Files

Rule files are simply files that contain one or more rule blocks. They can be used to allow agents to load certain default behaviours.

A.5 External Abilities

In order to allow agents to interact with the world, they can use arbitrary java objects that implement the Ability interface (Fig. A.3).

The object Pred implements the generic methods, so that only the execute() and init(Term) method need to be programmed if the pred is extended. Note that the argument Term represents the whole predicate, therefore allowing for an arbitrary number of arguments (for the predicate) to be used. The arguments can be accessed using the get(Integer) method of Term.

The system provides two basic abilities, namely Print and Wait. The former class “listens” to print/1 and writes its argument to standard out, while the latter (listening to wait(arg1,arg2)) waits for arg1 number of milliseconds before sending the Term arg2 to the InBox of the agent.
**Figure A.3.** The Ability interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor(Agent, String)</td>
<td>expects the agent and the name of the predicate</td>
</tr>
<tr>
<td>Constructor(Agent, String, Term)</td>
<td>ditto, plus a Term containing the arguments</td>
</tr>
<tr>
<td>String getName()</td>
<td>returns the name of the predicate that the object represents</td>
</tr>
<tr>
<td>void init(Term)</td>
<td>initialises the object with the arguments of the currently true predicate</td>
</tr>
<tr>
<td>Term execute()</td>
<td>The actual execution of the object</td>
</tr>
<tr>
<td>void reset()</td>
<td>sets all arguments to zero</td>
</tr>
<tr>
<td>Term getTerm()</td>
<td>returns the arguments of the predicate</td>
</tr>
<tr>
<td>Object clone()</td>
<td>needed to ensure that concurrent executions of the object are possible</td>
</tr>
</tbody>
</table>

### A.6 Internal Predicates

Structured METATEM knows a number of predicates that have internal side effects. Most of them are calling low-level methods that manipulate the environment of the agent, though the last few are mainly for debugging purposes.

**send/2** takes two arguments. The first should be a set-expression, and the second a Term that is sent to the agents specified in the set-expression.

**doAddToContent/1** adds the argument to the *Content* set of the agent. The argument can be a set-expression.

**doAddToContext/1** adds the argument to the *Context* set of the agent. The argument can be a set-expression.

**doAddToDirect/1** adds the argument to the *Known* set of the agent. The argument can be a set-expression.
doRemoveFromContent/1 removes the argument from Content and moves them to the Known set. An error is displayed if the agent is not known.

doRemoveFromContext/1 removes the argument from Context and moves them to the Known set. An error is displayed if the agent is not known.

doRemoveFromDirect/1 removes the argument from Known. An error is displayed if the agent is not known.

doRemove/1 removes the argument from either Content or Context and moves them to the Known set. An error is displayed if the agent is not known.

doClone/2 creates a new agent. First argument is the name of the new agent, and second a predicate set with as arguments predicates that will be translated into start rules.

stop/0 causes the agent to stop its execution.

prefer/2 can be used to re-order eventualities. At the moment, it moves arg1 directly before arg2, if arg2 was “before” arg1, and does nothing otherwise.

equals/2 will only be true if the two arguments are equal. equals/2 is examined at creation of a state, and unequal arguments will cause the system to immediately backtrack and try another set of possible predicates.

getContent/0 sends a predicate getContent/1 to the agent; the argument contains a list of the agents currently in the set /content.

getContext/0 sends a predicate getContext/1 to the agent; the argument contains a list of the agents currently in the set Content.
**getKnown/0**  sends a predicate `getKnownAgents/1` to the agent; the argument contains a list of the agents currently in the set `Content`.

**setDebug/1**  takes as argument either "true" or "false" and toggles the debug output accordingly.

**setStateDebug/1**  takes as argument either "true" or "false" and toggles the state debug output accordingly.

### A.7 Default Constructs

In order to facilitate the use of Structured METATEM, a rule file with some default behaviours can be used. It is called `default.rules`, and contains the following behaviours:

**sendAll(SetExpression, Message)**  can be used to recursively forward a message to a given set. Typical examples for sets would be “content union context” to address all agents within the system. Note that the implementation of this behaviour does not guarantee that agents will receive the message at most once. It is guaranteed however that no message will be re-sent forever.

**addToContent($Self,Agent)**  initiates a simple protocol that ensures that both the adding as well as the added agent actually are aware of the fact that they are adding/added (that is, keep their databases consistent). Similar rules exist for `addToContext/2`, `addToKnownAgents/2`.

**remove($Self,Agent)**  initiates a simple protocol that ensures that all parties are aware of the removal of an agent.

---

2Note that `$Self` is just used as an example for a variable.
goUp(Agent, $Self, Target) is a simple protocol (initiated by $Self that moves Agent to Target, where Target is in the Context of the moving agent. Similar rules exist for GoInto/3 (where Target would be in the Content, moveUp/3, and moveInto/3. Move?? is initiated by the agent planning to move, while go?? is initiated by another agent.

move(Sender, MovingAgent, ReceivingAgent) is a more generic version of the behaviours above.
Appendix B

Code

This chapter contains the full code of the multi-agent examples that we have discussed above for reference.

B.1 Default File

/**
   * sendAll(SetExpression S, Predicate Message)
   * sendAll(Set, Message) sends a message recursively through the
   * space defined by Set. Typical values would be {content} or
   * {content UNION context}.
   * Note the third line which ensures that send-messages are forwarded
   */
   sendAll: {
     sendAll(Set,Message) => NEXT send(Set, Message).
     sendAll(Set,Message) => NEXT send(Set, sendAll({Set WITHOUT $SELF}, Message)).
     receive(send(Set, Message)) => NEXT send(Set, Message).
   }

/**
   * receiveAll makes a very trusting agent. All received messages are
   * believed!
   */
   receiveAll: {
     receive(X) => NEXT X .
Code

/**
 * addToContent(Agent A, Agent B) /addToContext(Agent A, Agent B)
 * upon receiving addToContent, Agent a is added to the content. Then
 * a confirmation message "addedToContext($SELF)" is sent to this
 * agent (who is already reachable through content).
 * addedToContent(Agent a)
 * upon receiving this message, Agent a is added to the Context.
 * NOTE: This is a very simple way of dealing with addToContent, one
 * could e.g. make sure that only agents that were receivers of
 * addToContent could be added to Context using addedToContext.
 * addToContext works accordingly.
 */
addToContent: {
    addToContent($SELF,Sender) => NEXT doAddToContent(Sender).
    addToContent($SELF,Sender) => NEXT
        send(Sender, addedToContent($SELF,Sender)).
    addedToContent(Sender,$Self) => NEXT doAddToContext(Sender).
}
addToContext: {
    addToContext($Self, Sender) => NEXT doAddToContext(Sender).
    addToContext($Self, Sender) => NEXT
        send(Sender, addedToContext($SELF,Sender)).
    addedToContext(Sender,$Self) => NEXT doAddToContent(Sender).
}
addToKnownAgents: {
    addToKnownAgents($Self,Sender) => NEXT doAddToKnownAgents(Sender).
    addToKnownAgents($Self, Sender) => NEXT
        send(Sender, addedToKnownAgents($SELF)).
    addedToKnownAgents(Sender, $Self) => NEXT doAddToKnownAgents(Sender).
}
remove: {
    removeAgent($Self,Sender) => NEXT doRemove(Sender).
}
removeAgent($Self, Sender) => NEXT
    send(Sender, removed($Self, Sender)).
removed($Self, $SELF) => NEXT doRemove($Self).
removeFromContent($Self, Sender) => NEXT
doRemoveFromContent(Sender).
removeFromContent($Self, Sender) => NEXT
    send(Sender, removed($Self, Sender)).
removeFromContext($Self, Sender) => NEXT
doRemoveFromContext(Sender).
removeFromContext($Self, Sender) => NEXT
    send(Sender, removed($Self, Sender)).
removeFromKnown($Self, Sender) => NEXT
doRemoveFromKnown(Sender).
removeFromKnown($Self, Sender) => NEXT
    send(Sender, removed($Self, Sender)).
}
/**
* goUP (Sender, MovingAgent, TargetAgent)
* *
* movingAgent is the agent making the movement, to the
* Content of TargetAgent
*/
goUp: {
goUp(Sender, $Self, Target) => NEXT
    send(Sender, addToContent(Target, $Self)).
goUp(Sender, $Self, Target) => NEXT
    send(Sender, removeFromContent(Sender, $Self)).
}
/**
* goInto(Sender, MovingAgent, fromWhere)
* *
* goInto is defined to move from a certain context into the context
* of the sending agent. Note that fromWhere can be an agent,
* "context" or a set. At the moment however, sets cannot contain
* variables!
*/
goInto: {
goInto(Sender, $Self, From) => NEXT
    send(Sender, addToContent(Sender, $Self)).
goInto(Sender, $Self, From) => NEXT
    send(From, removeFromContext($Self, From)).
goInto(Sender, $Self, From) => NEXT
    send(Sender, removeFromContext(Sender,$Self)).
}

/**
** move(Sender, Agent_to_move, Agent_to_move_to)
*  
*/
move: {
    move(Sender, $Self, Target) => NEXT
        addToContext($Self, Target).
    move(Sender, $Self, Target) => NEXT
        removeFromContext($Self, Sender).
}

/**
** moveUp(MovingAgent,Context,Target)
*  
** moveUp is initiated by the agent planning to move up. It send the
** message with 2nd argument being the agent (or set of agents) that
** he wants to move up through to the 3rd argument, which again can
** be one agent, or a set (or just context).
*  
*/
moveUp: {
    moveUp($Self, ContextAgent, Target) => NEXT
        send(ContextAgent,send(Target,addToContent(Target,$Self))).
    moveUp($Self, ContextAgent, Target) => NEXT
        send(ContextAgent,removeFromContent(ContextAgent, $Self)).
}

moveInto: {
    moveInto($Self, FromGroup, TargetAgent) => NEXT
        removeFromContext($Self,FromGroup).
    moveInto($Self, FromGroup, TargetAgent) => NEXT
        removeFromContext($Self,TargetAgent).
    moveInto($Self, FromGroup, TargetAgent) => NEXT
        addToContext($Self,TargetAgent).
}
B.2 Coffee Example Code

B.2.1 Environment File

//coffee robot example
agent coffee: Agent/c_robot.agent;
agent room1: Agent/c_room1.agent;
agent room2: Agent/c_room2.agent;
agent room3: Agent/c_room3.agent;
agent room4: Agent/c_room4.agent;
agent corr: Agent/c_corr.agent;
agent user: Agent/c_user.agent;

coffee {
    context: room1;
    known: room2, room3, room4, corr, user;
}

room1 {
    content: coffee;
    context: corr;
}

room2 {
    context: corr;
}

room3 {
    context: corr;
}

room4 {
    content: user;
    context: corr;
}

corr {
    content: room2, room3, room1, room4;
}

user {
    context: room4;
}
B.2.2 Robot Agent

Note that in order to allow for the simple definition of multiple robots, all rules are moved into a rule files which is loaded via the `include` command.

Robot Agent File

```plaintext
name robot;

include Agent/c_default.rules;
include Agent/c_robot.rules;

ability print: Agent/Print;
ability wait: Agent/Wait;
```

Robot Rules File

```plaintext
receiving_request :{
  receive(needCoffee(Room, User, ID)) => NEXT
  delivering(ID).
  receive(needCoffee(Room,needCoffee(NewRoom,User,ID), ID)) => NEXT
  send(NewRoom,addToContent(NewRoom,$Self,)).
  receive(needCoffee(Room,needCoffee(NewRoom,User,ID), ID)) => NEXT
  send(Room,removeFromContent(Room,$Self)).
  receive(needCoffee(Room, User, ID)) => NEXT
    nextMove(User).
  receive(needCoffee(Room, User, ID)) => NEXT
    moving().
}

keep_walking: {
  nextMove(User), not(receive(addedToContent(Room,$Self))) => NEXT
  nextMove(User).
  nextMove(needCoffee(Room,User,ID)) => NEXT
    receive(needCoffee(Room,User,ID)).
  nextMove(needCoffee(User,ID)) => NEXT
    print(broughtCoffee(User,ID)).
}
```

B.2.3 Room Agent

Note that in order to allow for the simple definition of multiple rooms, all rules are moved into a rule files which is loaded via the `include` command.

Corridor agents use the same code base as room agents.
Example Room Agent File

```plaintext
name room1;

include Agent/c_default.rules;
include Agent/c_room.rules;

statedebug false;

ability print: Agent/Print;
ability wait: Agent/Wait;
```

Room Rules File

```plaintext
forwardrequest: {
    receive(needCoffee(User,ID)), not(sentRequest(ID)) => NEXT
    send({content UNION context},
         needCoffee($Self,needCoffee(User, ID),ID)).
    receive(needCoffee(Room,User,ID)), not(sentRequest(ID)) => NEXT
    send({content UNION context},
         needCoffee($Self,needCoffee(Room, User, ID),ID)).
    receive(needCoffee(User,ID)), not(sentRequest(ID)) => NEXT
         sentRequest(ID).
    receive(needCoffee(User,Room,ID)), not(sentRequest(ID)) => NEXT
         sentRequest(ID).
}

remember: {
    sentRequest(ID), not(receive(delivering(ID))) => NEXT
         sentRequest(ID).
}
```

B.2.4 User Agent

Note that in order to allow for the simple definition of multiple rooms, all rules are moved into a rule files which is loaded via the `include` command.

User Agent File

```plaintext
name user;

include Agent/c_default.rules;
include Agent/c_user.rules;
//statedebug true;
```
ability print: Agent/Print;
ability wait: Agent/Wait;

begin: {
    START => go().
    go() => NEXT wait(2000, askforCoffee()).
    receive(askforCoffee()), inRoom(Room) => NEXT
    send(context, needCoffee(Room,$Self)).
}

User Rules File

whereami: {
    START => whereami().
    whereami() => NEXT send(context, whatRoom($Self)).
    receive(room(Room)) => NEXT inRoom(Room).
    inRoom(Room) => NEXT inRoom(Room).
}

B.2.5 Message Exchange

user(0)(1)(1): receive(askforCoffee())
user(0)(1)(1): send(context(), needCoffee(user(0)(1)(1), one()))
corridor(4)(0)(0): receive(needCoffee(room(4)(1)(1)(0), needCoffee(user(0)(1)(1), one()), one()))
room2(0)(1)(0): receive(needCoffee(corridor(4)(0)(0), needCoffee(room(4)(1)(1)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
room2(0)(1)(0): send(union(set(content()), set(context())), needCoffee(room2(0)(1)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
room3(0)(1)(0): receive(needCoffee(corridor(4)(0)(0), needCoffee(room(4)(1)(1)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
corridor(4)(0)(0): send(union(set(content()), set(context())), needCoffee(corridor(4)(0)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
room3(0)(1)(0): send(union(set(content()), set(context())), needCoffee(room3(0)(1)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
room1(1)(1)(0): receive(needCoffee(corridor(4)(0)(0), needCoffee(room1(1)(1)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
user(0)(1)(1): receive(needCoffee(room1(1)(1)(0), needCoffee(user(0)(1)(1), one()), one()))
robot(0)(1)(5): receive(removeFromContent(room1(1)(1)(0), robot(0)(1)(5)))
corridor(4)(0)(0): receive(needCoffee(room2(0)(1)(0), needCoffee(corridor(4)(0)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
corridor(4)(0)(0): receive(needCoffee(room3(0)(1)(0), needCoffee(corridor(4)(0)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
corridor(4)(0)(0): receive(needCoffee(room1(1)(1)(0), needCoffee(corridor(4)(0)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
corridor(4)(0)(0): send(union(set(content()), set(context())), needCoffee(corridor(4)(0)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
room1(1)(1)(0): send(union(set(content()), set(context())), needCoffee(room1(1)(1)(0),
    needCoffee(user(0)(1)(1), one()), one()), one()))
robot(0)(1)(5): send(corridor(4)(0)(0), addToContent(corridor(4)(0)(0), robot(0)(1)(5)))
robot(0)(1)(5): send(room1(1)(1)(0), removeFromContent(room1(1)(1)(0), robot(0)(1)(5)))
robot(0)(1)(5): nextMove(needCoffee(corridor(4)(0)(0),
    needCoffee(room1(1)(1)(0), needCoffee(user(0)(1)(1), one()), one()), one()))
room1(0)(1)(1): send(robot(0)(1)(5), removed(room1(0)(1)(1), robot(0)(1)(5)))
robot(0)(1)(5): receive(needCoffee(corridor(4)(0)(0),
    needCoffee(room1(1)(1)(0), needCoffee(user(0)(1)(1), one()), one()), one()))
B.3 Museum Example Code

B.3.1 Environment File

//Active Museum Physical Setup2

agent room1: Agent/room11.agent;
agent room2: Agent/room12.agent;
agent room3: Agent/room13.agent;
agent room4: Agent/room14.agent;
agent sensor1: Agent/sensor11.agent;
agent sensor2: Agent/sensor12.agent;
agent sensor3: Agent/sensor13.agent;
agent sensor4: Agent/sensor14.agent;
agent visitor1: Agent/visitor1.agent;
agent visitor2: Agent/visitor2.agent;
agent museum: Agent/museum.agent;

museum {
    content: sensor1,sensor2,sensor3,sensor4;
}

room1 {
    content: sensor1;
}
room2 {
    content: sensor1;
}


B.3.2 Room Agent

Note that in order to allow for the simple definition of multiple rooms, all rules are moved into a rule files which is loaded via the include command.

Example Room Agent

name room1;
include Agent/defaultMuseum.rules;
include Agent/s_room.rules;
ability print: Agent/Print;
ability wait: Agent/Wait;

Room Rules File

sendExhibits: {
    receive(looking(Visitor,$Self)) => NEXT
    send(content,whatExhibit($Self,Visitor)).
    receive(exhibit(Exhibit,$Self,Visitor)) => NEXT
    send(Visitor,canSee(Visitor,$Self,Exhibit)).
}

room: {
    receive(whatRoom(X)) => NEXT send(X,room($Self)).
}

init: {
    movevisitor() => NEXT
    send(content, moveTo($visitor1,$Self,$museum)).
    receive(moveTo(Visitor,From,To))=> NEXT
    send(Visitor, moveTo(Visitor,From,To)).
}

B.3.3 Visitor Agent

Note that in order to allow for the simple definition of multiple rooms, all rules are moved into a rule files which is loaded via the include command.

Example Visitor Agent

name visitor1;
include Agent/defaultMuseum.rules;
include Agent/s_visitor1.rules;

//debug true;
//statedebug true;

ability print: Agent/Print;
ability wait: Agent/Wait;

Visitor Rules File

whereAmI: {
    START => do().
    do() => NEXT send(context, whatRoom($Self)).
receive(room(Room)) => NEXT inRoom(Room).
inRoom(Room), not(moving()) => NEXT inRoom(Room).
inRoom(Room), not(receive(moveTo($Self,X))) => NEXT inRoom(Room).
moving(), moveTo($Self,X) => NEXT inRoom(X).
}

cleanup: {
  receive(moveTo($Self,X)), inRoom(Room),
  canSee($Self, Room,Exhibit) => NEXT seen(Exhibit).
}

movement: {
  receive(moveTo($Self,X)),inRoom(Room) => NEXT
  send(Room,removeAgent(Room,$Self)).
  receive(moveTo($Self,X)) => NEXT send(X,addToContent(X,$Self)).
}

exhibits: {
  addedToContent(Room,$Self) => NEXT lookAround(Room).
  lookAround(Room) => NEXT send(context, looking($Self,Room)).
  receive(canSee($Self,Room,Exhibit)) => NEXT
    send(context,canSee($Self,Room,Exhibit)).
}

waiting: {
  canSee($Self,Room,Exhibit) => SOMETIME goLooking(Exhibit).
  canSee($Self,Room,Exhibit) => NEXT not(goLooking(Exhibit)).
  canSee($Self,Room,Exhibit) => NEXT wait(9000,startLooking()).

  not(goLooking(Exhibit)),not(startLooking()) => NEXT
    not(goLooking(Exhibit)).
  receive(startLooking()) => NEXT startLooking().
}

preference: {
  receive(prefer(Room,Exhibit1,Exhibit2)) =>
    NEXT prefer(goLooking(Exhibit1),goLooking(Exhibit2)).
}

looking: {
  goLooking(X),goLooking(Y) => NEXT equals(X,Y).
  goLooking(Exhibit), not(seen(Exhibit)) => NEXT lookAt(Exhibit).
B.3 Museum Example Code

```plaintext
goLooking(Exhibit), discard(Exhibit) => NEXT seen(Exhibit).
lookAt(Exhibit) => NEXT seen(Exhibit).
seen(Exhibit) => NEXT not(lookAt(Exhibit)).
seen(X) => NEXT seen(X).
}

talk: {
    seen(X) => NEXT print(seen(X)).
    discard(X) => NEXT print(discard(X)).
    lookAt(X) => NEXT print(lookAt(X)).
}

eclude: {
    receive(discard(X)) => NEXT discard(X).
    discard(X) => NEXT discard(X).
}

B.3.4 Example Artist Agent

name artistgroup;
include Agent/defaultMuseum.rules;

ability print: Agent/Print;
ability wait: Agent/Wait;

preferences: {
    START => go().
go() => NEXT prefer($room1,$exhibit1,$exhibit2).
go() => NEXT prefer($room1,$exhibit3,$exhibit2).
go() => NEXT prefer($room1,$exhibit1,$exhibit3).
go() => NEXT prefer($room2,$exhibit4,$exhibit6).
go() => NEXT prefer($room2,$exhibit5,$exhibit4).
go() => NEXT prefer($room2,$exhibit5,$exhibit6).

    prefer(X,Y,Z) => NEXT prefer(X,Y,Z).
}

request: {
    receive(canSee(Visitor,Room,Exhibit)) => NEXT
        canSee(Visitor,Room,Exhibit).
    canSee(User,Room,Exhibit), prefer(Room, Ex1, Ex2) => NEXT
        send(content,prefer(Room,Ex1,Ex2)).
}
```
## List of Figures

2.1 The possibility and necessity operator .................................. 14
2.2 BOID control loop .............................................................. 23

3.1 Temporal Operators .............................................................. 41
3.2 Rules in SNF ................................................................. 43
3.3 Simple formulae in SNF form ................................................. 44
3.4 Possible models for example formulae ....................................... 45
3.5 A graphical representation of belief contexts .............................. 47
3.6 Rules in SNF\textsubscript{BB} .................................................. 50
3.7 General algorithm for executing METATEM ............................... 55
3.8 Outline Algorithm for Rule Interpretation .................................. 56
3.9 Prolog algorithm for METATEM .............................................. 59
3.10 Example of an agent definition file ........................................ 64

5.1 Different views of the system structure .................................... 109
5.2 How agents within a group communicate ................................... 111
5.3 Code block implementing \texttt{addToContent} ................................ 118
5.4 Graphical representation of agent movements ............................. 118
5.5 Code snippet showing a simple implementation of the \texttt{goUp} action 119
5.6 Manipulating the agent space .................................................. 120
5.7 Examples of using structure .................................................. 121
5.8 Using \textit{Content} ............................................................ 123
5.9 Using external group ........................................................... 124
5.10 Using internal group ........................................................ 125
5.11 Different properties that groups can have ................................ 126
5.12 Graphical representation of \textit{Known} set ................................. 130

6.1 Example Program to show backtracking ..................................... 146
6.2 Graphical representation of execution trace ............................... 146
6.3 Execution trace of program shown in Fig. 6.1 ................................ 148
6.4 Layout of the football pitch .................................................. 149
6.5 Example code for bounded belief ............................................ 150
6.6 Output of run with belief depth 2 ......................................... 151
6.7 Output of run with belief depth 0 ......................................... 151
6.8 Example code for deliberation ................................................. 154
6.9 Example output of restaurant agent ........................................ 155
7.1 Layout of coffee example .................................................. 158
7.2 An example of an environment definition ................................. 158
7.3 An example file for a room agent ....................................... 159
7.4 An example file for a coffee robot ..................................... 160
7.5 Physical Structure of Museum Example .................................. 163
7.6 Organisational Structure of Museum Example ............................ 164
7.7 Physical space rules of both Visitor and Room Agents ................. 165
7.8 Organisational space rules of Interest Group Agent ................. 165
7.9 Organisational space rules of Visitor Agent ............................. 167
7.10 Messages sent when moving to a room ................................ 168
7.11 Message exchange with interest groups ................................ 170
8.1 Example of a place graph .................................................. 182
8.2 Example of a link graph .................................................... 183
8.3 Example of a bigraph ....................................................... 184
8.4 A reaction rule for agents .................................................. 185
8.5 Agents representing place graph ....................................... 188
8.6 Agents representing link graph ......................................... 189
8.7 Agents representing bigraph ............................................. 191
A.1 The Structure of an environment file .................................. 223
A.2 List of header commands for agent definition files .................. 223
A.3 The Ability interface ...................................................... 226
# List of Definitions

<table>
<thead>
<tr>
<th></th>
<th>Definition</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Constants</td>
<td>66</td>
</tr>
<tr>
<td>2</td>
<td>AgentNames</td>
<td>66</td>
</tr>
<tr>
<td>3</td>
<td>Terms</td>
<td>66</td>
</tr>
<tr>
<td>4</td>
<td>Predicates</td>
<td>67</td>
</tr>
<tr>
<td>5</td>
<td>Rules</td>
<td>67</td>
</tr>
<tr>
<td>6</td>
<td>Beliefs</td>
<td>67</td>
</tr>
<tr>
<td>7</td>
<td>Agent Description</td>
<td>67</td>
</tr>
<tr>
<td>8</td>
<td>States</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>Model</td>
<td>68</td>
</tr>
<tr>
<td>10</td>
<td>Semantic Function</td>
<td>68</td>
</tr>
<tr>
<td>11</td>
<td>History Function</td>
<td>69</td>
</tr>
<tr>
<td>12</td>
<td>Ordering</td>
<td>70</td>
</tr>
<tr>
<td>13</td>
<td>State Function</td>
<td>72</td>
</tr>
<tr>
<td>14</td>
<td>Choice Function</td>
<td>74</td>
</tr>
<tr>
<td>15</td>
<td>Merging Partial States</td>
<td>76</td>
</tr>
<tr>
<td>16</td>
<td>Single Rule Interpretation</td>
<td>78</td>
</tr>
<tr>
<td>17</td>
<td>Applying Literals to Rules</td>
<td>79</td>
</tr>
<tr>
<td>18</td>
<td>Extracting States from Rules</td>
<td>80</td>
</tr>
<tr>
<td>19</td>
<td>State Function with Belief</td>
<td>83</td>
</tr>
<tr>
<td>20</td>
<td>Belief Function</td>
<td>83</td>
</tr>
<tr>
<td>21</td>
<td>Message</td>
<td>132</td>
</tr>
<tr>
<td>22</td>
<td>Environment</td>
<td>132</td>
</tr>
<tr>
<td>23</td>
<td>States</td>
<td>132</td>
</tr>
<tr>
<td>24</td>
<td>Agent Space</td>
<td>132</td>
</tr>
<tr>
<td>25</td>
<td>Multiple Agent Model</td>
<td>133</td>
</tr>
<tr>
<td>26</td>
<td>Semantic Function</td>
<td>133</td>
</tr>
<tr>
<td>27</td>
<td>Multiple History Function</td>
<td>133</td>
</tr>
<tr>
<td>28</td>
<td>Global State Transition</td>
<td>134</td>
</tr>
<tr>
<td>29</td>
<td>Choice Function for Multiple Agents</td>
<td>136</td>
</tr>
<tr>
<td>30</td>
<td>State Transition for Multiple Agents</td>
<td>137</td>
</tr>
<tr>
<td>31</td>
<td>Interpreting Set Expressions</td>
<td>139</td>
</tr>
<tr>
<td>32</td>
<td>Place Graph</td>
<td>182</td>
</tr>
<tr>
<td>33</td>
<td>Link Graph</td>
<td>183</td>
</tr>
<tr>
<td>34</td>
<td>Bigraph</td>
<td>183</td>
</tr>
</tbody>
</table>
Index

3APL, 31–33

AALAADIN, 103, 177
Abilities, 27, 45, 49, 53
Agent0, 18
AgentSpeak(L), 18, 32, 102
ALIAS, 37
Ambient Calculus, 180, 185
Auctions, 101
Axiom
4, 46, 49
5, 49
D, 17, 49
fun, 49
K, 17
Successor State, 25

Backtracking, 58, 145
BDI, 6, 15–20, 42, 45, 97, 102, 178
Behavioural AI, 5
Belief, 15
Bigraphs, 180–193, 196
BOID, 20
Bratman, M. E., 6, 8, 15, 16
Broersen, J., 21
Brooks, R., 5
Busetta, P., 93, 110

Cohen, P. R., 16, 96–98
Communication, 93, 110, 129
Concurrency, 91
Confidence, 45, 46, 127
Congolog, see Golog
Content, 51, 61, 65, 109, 117, 122, 130, 186, 197
Context, 51, 61, 65, 109, 117, 122, 130, 186, 197
Contract Net, 100
CTL*, see Branching Time Temporal Logic
Dennett, D., 6
dMARS, 18, 102, 178
Dribble, 31, 33
Ferber, J., 103, 177
File
Agent Definition, 61, 63, 65, 223
Environment, 61, 65, 129, 222
Rules, 61, 65, 225
FIPA, 96
Fisher, M., 43, 45, 51, 52, 54, 60
Fluent, 24
Georgeff, M. P., 8, 16, 18, 178
Ghidini, C., 45, 60
GOAL, 31, 33
Goldblatt, R., 30
Golog, 26, 32
Grosz, B., 98
Groups, 103, 108, 121, 171, 177
Gutknecht, O., 177
Herbrand Model, 35
Hindriks, K., 31
Hintikka, J., 14
History Level, 69, 133
van der Hoek, W., 27, 28
Holonic Agents, 99, 175
IMPACT, 104
Interrap, 19
JACK, 102, 179
JASON, 18
Joint Commitment, 97
Joint Intention Theory, 97, 103, 177

Kakoudakis, T., 51
Karo, 8, 27–31, 45
Known, 130, 179, 186
KQLM, 96
Kraus, S., 98
Kripke Model, see Possible World Semantics
Kripke, S., 14

LAILA, 38
Lesser, V., 178
Levesque, H. J., 16, 96, 97
van Linder, B., 27
Link Graph, 181, 182

Logic
Branching Time Temporal, 16
Default, 20–23
Dynamic, 96
K, 48
KD45, 15, 17, 46
Modal, 13–15
Multi Context, 45
of Capabilities, 27
Predicate, 34
S4, 14
S5, 14, 15
Temporal, 39, 50
Temporal Bounded Belief, 47

Logic Programming, 33
LoudVoice, 162

MadKit, 103, 177
Magenta, 54
Match Maker, 100
METATEM, 39, 44, 50, 59, 61, 107, 131, 195
Concurrent, 1, 50
Meyer, J. J., 27
Milner, R., 180, 181, 184, 191
MINERVA, 37
Moore, R. C., 8, 27
Newell, A., 4, 5
Norms, 20
Operational Semantics, 19, 64, 131
Path Formula, 16
PEACH Project, 161
Physical symbol system hypothesis, 4
π-Calculus, 93, 180
Place Graph, 181
Plotkin, G. D., 19, 65
Possible World Semantics, 8, 14, 16, 29, 198
prefer, 53, 152
PRS, 18, 102

Rao, A., 8, 16, 18, 178
Reiter, R., 20, 21
Reputation, 101
Roles, 99, 103, 177, 180
Rule Block, 61
Rules Level, 75

Searle, J. R., 99
Separated Normal Form(SNF), 43, 49
SharedPlan Model, 98
Simon, H., 4, 5
Situation Calculus, 23–26
Speech Act, 95
start, 41
State Formula, 16
State Level, 69, 134
STEAM, 103
Sub-Rule Level, 78

Tambe, M., 103
Team, 180
TeamCore, 103
TL, see Temporal Logic
Threads, 92
Tidhar, G., 98, 177, 178
TLBB, see Temporal Logic with Bounded Belief

Z, 19
Zhang, H., 178