Communication in Goal Oriented Agents

Master's Thesis

Wouter de Vries

Communication in Goal Oriented Agents

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

MEDIA AND KNOWLEDGE ENGINEERING

by

Wouter Adelbert de Vries
born in Leiderdorp, the Netherlands

Man-Machine Interaction Research Group
Department of Mediamatics
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

© 2010 W.A. de Vries.

Communication in Goal Oriented Agents

Author:	Wouter de Vries
Student id:	1015842
Email:	`w.a.devries@student.tudelft.nl`

Abstract

In this thesis the agent programming language GOAL is extended with communication. For GOAL we pursue an implementation that has a well-founded theory as well as providing pragmatic programming constructs for the programmer. Existing technologies are reviewed to explore their approaches. Many of these technologies have Speech-act theory as their theoretic base, or build upon other technologies that do so. This led to communication frameworks having vast and unclear performative sets and lacking formal semantics.

This thesis goes back to the core of communication by regarding communication from a linguistic perspective. This approach inspires syntactical representation of communication constructs and also leads the way to specifying a formal semantics for those communication constructs. This semantics does not have a receiver of a message refer directly to the mental state of the sender, but rather specifies how a *model* of that mental state can be deduced from the communication. These mental models are implemented as language constructs which the agent programmer can use to have the agents reason about the beliefs and goals of other agents.

Finally the necessary middleware elements are implemented into the GOAL interpreter to allow a system of multiple agents to be distributed across multiple platforms or hosts.

Thesis Committee:

Chair:	Prof. Dr. C.M. Jonker, Faculty EEMCS, TU Delft
University supervisor:	Dr. K.V. Hindriks, Faculty EEMCS, TU Delft
Committee Member:	Dr. M.M. Dastani, Faculty of Science, Utrecht University

Preface

As I start writing this last part of this document I cannot help but reflect on the past period, and especially the process of doing research and writing this thesis. In much the same way as concerning the subject matter I had moments where I couldn't see the forest for the trees, these moments where equally present in the personal attitude towards the process. It feels like walking a path through a hilly landscape, having a general idea of the direction to take towards the goal. Most of the time you see the path in front of you, but sometimes you lose the path when walking in a valley, and you need to climb some hill to reorient yourself.

I have seen a lot of valleys and hilltops during my graduation period, but I consider myself lucky to have learned the landscape. Not only have my 'pathfinding' skills been improved, I can also take home the experience of making such a journey, and seeing it through till the end, even when the valleys and hills sometimes seemed to stretch as far as I could see. Were it not for some essential guides that repeatedly showed me where the path was, it might have been hard to ever get to the goal.

There have been many guides in many forms, all of which have my sincere gratitude, but a few I would like to mention;

First of all, Koen I thank for the huge patience in supervising my graduation. Whenever I once again disappeared into some valley only to reappear some indefinite time later, we could always continue where I left off. At times it took some persuasion to keep me away from getting too philosophical or getting lost in the details.

Tijmen, thanks for the company in our graduation room. The essential coffee breaks and chats about whatnot were the kind of distraction that actually fuel the work spirit.

Finally I would like to thank Noor, for always being there for me. She always knew why I was lost and gently showed me how to find the path again.

Wouter de Vries
Delft, the Netherlands
May 8, 2010

Contents

List of Figures

Chapter 1

Introduction

One of the properties characteristically ascribed to intelligent agents is their ability to interact. Another is that they should be autonomous. One form of interaction between agents that respects autonomy is *communication*. Communication allows agents in a multi-agent system to perform better (i.e. more efficient, solving more problems) than they would alone or together without interacting.

Using agents as a programming paradigm requires a way to convert these agent concepts into software. Agent programming languages aim to provide the bridge between concept and application. Over the past decades, increased interest for agent technologies has given rise to development of agent programming languages (APLs) and frameworks. While some of these APLs focus on the formal aspects of agents ([7],[1]), others aim to provide a reasoning framework ([2]). GOAL is a Goal Oriented Agent Language in which the agent is specified declaratively in terms of *beliefs* and *goals*. GOAL is currently being developed at the Delft University of Technology.

In this thesis I extend GOAL with the ability for agents to communicate. To achieve this, the following questions are investigated:

1. What is minimally required to allow communication between agents?
2. How easy can communication be used or applied in GOAL programs?

As the name suggests, GOAL agents are specified declaratively in terms of their goals. When communicating, it would be nice to do so based on the contents of their mental states. From GOAL's perspective, we would like to observe the following criteria for success for designing and specifying an ACL:

1. **The communication constructs should have a well-founded theory.** This includes a formal semantics of the communicative acts. The importance of such a theory lies in an agents program's verifiability.
2. **The distinction between beliefs and goals should persist in the communication.** GOAL is based on this distinction, and after receiving a message an agent should be able to determine whether the message conveyed a statement about the sending agent's beliefs or about its goals; the receiving agent should be able to distinguish informational from motivational content.
3. **Programming agents using the communication constructs should be pragmatic, and pose a small burden on the programmer.** An agent programmer should not have to introduce complexity or complicated constructs to make the agents communicate.

When the informational and motivational parts of a message can be distinguished this gives us an insight into the mental state of the sender. With these accumulated insights an agent can reason about the mental state of other agents, even if that reasoning is based on a *model* of the mental state of that other agent. In this thesis I introduce *mental models*. The concept, semantics and implementation are described and accompanied by code examples demonstrating their usage.

1.1 Examples of Multi-Agent Systems

In single-agent systems and -environments, communication is not really an issue, and such agents need not consider other agents or their influence on the environment and the agent's plans. In situations where more than one agent operates in the same environment, however, interaction between agents may become beneficial or necessary for the execution of the individual agent's plans. It might be that agents could carry out their plans individually, but if they cooperate, they could do it more efficiently. On the other hand, it could be that the presence of another agent with its own agenda will interfere with the agent's plan. Rosenschein and Zlotkin [14] identified 4 situations of two agents with respect to their relative goals:

1. The *symmetric cooperative* situation, in which both agents could execute their plans on their own, but would both benefit from cooperating on those tasks. They both welcome the other's presence.

2. The *symmetric compromise* situation, in which both agents would rather execute their plans on their own, but are forced to deal with each other in order to have their plans executed.

3. The *non-symmetric cooperative/compromise* situation, in which one of the agents benefits from the interaction, while the other is worse off interacting than working alone.

4. The *conflict* situation, in which the goals of the agents contradict each other, i.e. the goal states of both agents cannot be achieved simultaneously.

1.1.1 Rickety delivery

Suppose two package delivery agents, $\mathcal{A}_1$ and $\mathcal{A}_2$, each have a package to deliver by a certain time t, at a certain location l, which is just across a rickety rope bridge. Delivery agents can hold at most two packages at any time. Carrying packages a certain distance has a cost of c per package. The bridge is only strong enough to hold one agent at a time.

Time constraints Suppose that as the agents stand at the beginning of the bridge, there is no time left for both agents to cross the bridge consecutively. Obviously, one agent could carry both their packages across the bridge, thus achieving both their goal states (their package being at l by time t) for them. This situation is a *non-symmetric compromise* for the agent carrying the packages, and a *non-symmetric cooperation* for the agent not having to do work at all.

If the agents were to ignore each other, the first one to reach the bridge would cross it, deliver its package and reach its goal state. But then the other agent will not be able to do so, leading to a global utility that is sub-optimal.

The way the agents might interact could be as follows. Agent $\mathcal{A}_1$ meets agent $\mathcal{A}_2$ at the start of the bridge, who is about to cross it. Their individual plans for achieving their goals lead them to a deadlock. Agent $\mathcal{A}_1$ would like the other to help it out, but it doesn't know the other's goal or plan, so it can't figure out what kind of plan will help them both to their goals. $\mathcal{A}_1$ would like to inspect $\mathcal{A}_2$'s goal base. But, being autonomous agents, that is not directly possible. So, $\mathcal{A}_1$ *asks* $\mathcal{A}_2$ to share the contents of it's goal base, and any beliefs it might have. $\mathcal{A}_2$ responds that it's goal is to get it's package to the other side of the bridge, at time t. Also, it mentions that it is carrying one package. Using this information, $\mathcal{A}_2$ thinks of a plan, namely $\mathcal{A}_1$ taking both packages to the other side while doing nothing itself (saving energy), and *requests* $\mathcal{A}_1$ to do so. $\mathcal{A}_2$ *agrees* explicitly, and the *joint plan* is executed.

Weight constraints What if the two packages were sealed together in one container (which has no weight of its own). Either agent could carry the container to the other side, to achieve its goal, which has the side effect of achieving the other agent's goal as well. Without coordination, both agents will have their goals achieved, regardless of which agent executes the task. But they will both attempt to execute the task themselves, since they have no idea that the other agent has the same (sub)goal.

But suppose carrying the container costs $2c$, so if both agents would carry it together, cooperatively, it would each cost them only c, so both agents are better off together than expected alone. This is an example of a *symmetric cooperation*; both agents welcome the other's presence.

In this case, both agents would ask each other about their goals, and when they find out they have mutual (sub)goals, they construct a joint plan of carrying the same container together.

Conflict Suppose now that the time constraint does exist, that is, there is only time for one agent to cross the bridge, and that agents can carry at most one package at a time. In this case there is no sequence of actions that will bring about the combined goal state of both agents; the agents are in *conflict*. The best they can do by communicating about this situation is *negotiation*, but that topic goes beyond the scope of this document.

The rickety rope bridge examples have shown us that there are situations where cooperation is beneficial for all agents involved, or for some agents involved, or for the system of agents as a whole.

The communication in the above situations enabled the agents to come up with a joint plan that is better (i.e. has a higher global utility) than the sum of their individual plans. In order to come up with such a plan, agents must be aware of the goals of the other agent(s) involved.

1.1.2 Slotted Blocks World

Let's consider a simple domain in more detail. Consider a table with blocks on it, that can each be directly on the table, or on another block. There are a finite amount of slots these stacks of blocks can stand on. The state of the world can be described with a conjunction of predicates;

- CLEAR(A), meaning there is no block on A, where A can be a block or the table.
- ON(B, C), meaning block B is directly on top of C, where C can be a block or the table.
- AT(D, n), meaning block D is directly on the table in slot n.

There exists only two basic actions agents in this domain can perform:

- PICKUP(i), meaning that the agent picks up the block that is on the top of the stack in slot i. This can only be performed if slot i is not empty. The result is that the block is removed from the stack and the agent is now carrying it.
- PUTDOWN(i), meaning that the agent places the block it is holding on the stack (if any) in slot i.

The cost of one such operation is 1. Agents can carry at most one block at a time.

Typical start- and goal states are some arrangement of the blocks, usually all blocks being stacked in a specific order.

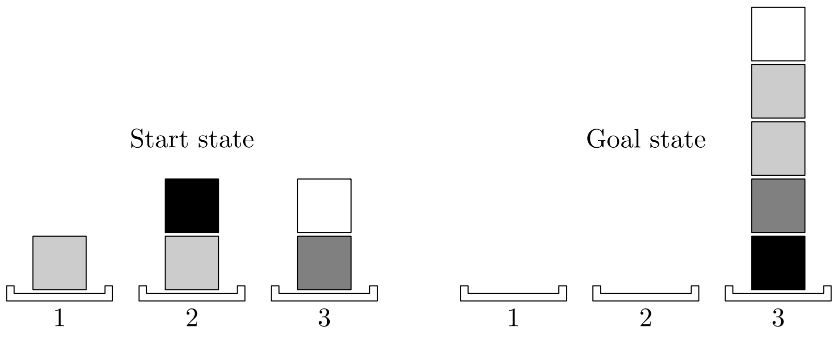

Figure 1.1: Example start- and goal state

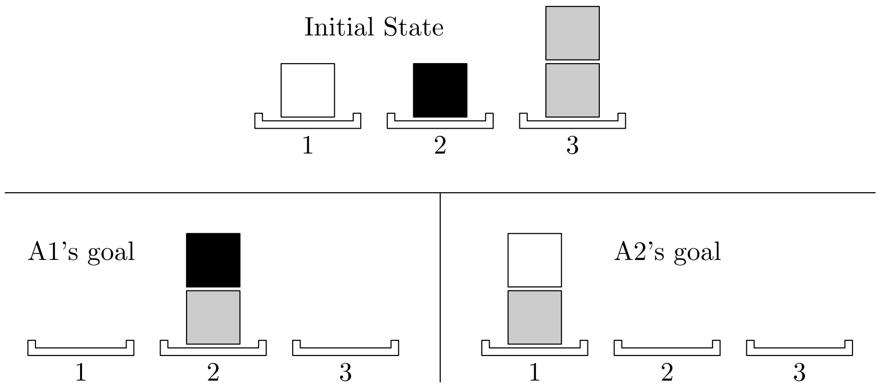

Figure 1.2: Initial- and goal states for two-agent Blocks World example problem

Figure 1.1 shows an example of such a slotted blocks world.

Now let's consider the following situation, illustrated by Figure 1.2. The initial state consists of a white block, a black block, and two stacked gray blocks. $\mathcal{A}_1$'s goal is to have the black block on a gray block on the table at slot 2, while $\mathcal{A}_2$'s goal is to have the white block on a gray block on the table at slot 1.

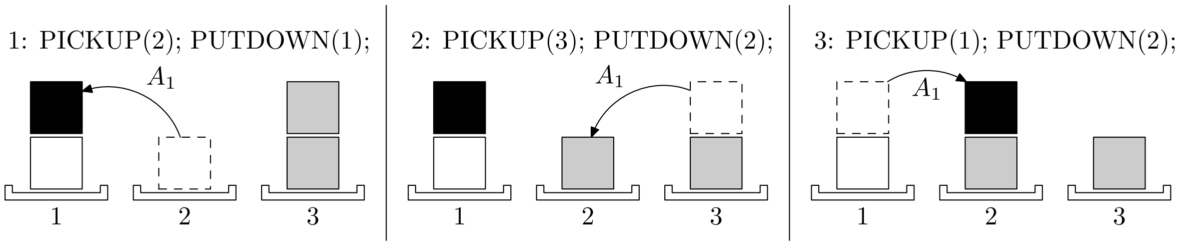
1: PICKUP(2); PUTDOWN(1);
A_1
1
2
3
2: PICKUP(3); PUTDOWN(2);
A_1
1
2
3
3: PICKUP(1); PUTDOWN(2);
A_1
1
2
3

Figure 1.3: Actions taken by $\mathcal{A}_1$ to achieve its goal

Suppose each agent operates on its own to achieve its goal. For $\mathcal{A}_1$, the associated actions are depicted in Figure 1.3. For $\mathcal{A}_2$, they are shown in Figure 1.4. In principle, these goals are not in

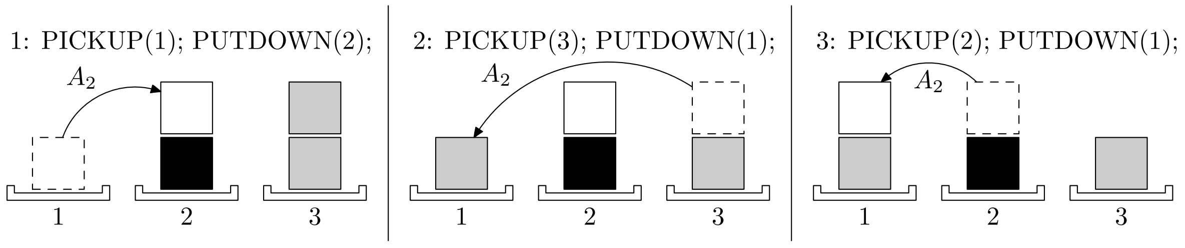
1: PICKUP(1); PUTDOWN(2);
2: PICKUP(3); PUTDOWN(1);
3: PICKUP(2); PUTDOWN(1);
A_2
A_2
A_2
1
2
3
1
2
3
1
2
3

Figure 1.4: Actions taken by $\mathcal{A}_2$ to achieve its goal

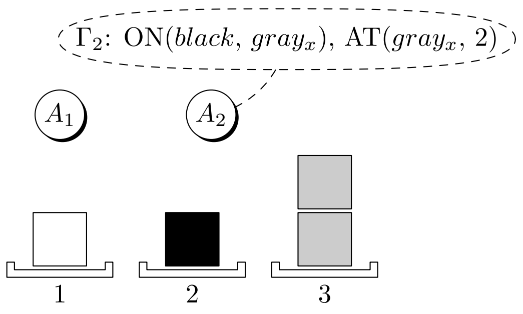

Figure 1.5: $\mathcal{A}_2$ informs $\mathcal{A}_1$ of its goal

	$\mathcal{A}_1$	$\mathcal{A}_2$
1		`inform(goal(`ON($black$,$gray_x$), AT($gray_x$, 2)`))`
2	`request(`PICKUP(1)`)`	
3		⟨ picks up white block at 1 ⟩
4	⟨ rearranges blocks to match Figure 1.7 ⟩	
5	`request(`PUTDOWN(1)`)`	
6		⟨ puts down the white block at 1 ⟩

Figure 1.6: Two agents performing a joint plan together

conflict, because there exists a (reachable) state in which both goals are satisfied (the union of the goal states in Figure 1.2). However, a problem does arise when the two agents perform their actions concurrently on the same environment: when $\mathcal{A}_1$ puts the black block on the white block, $\mathcal{A}_2$ is hindered in performing its first intended action, because the white block is no longer free. Removing the black block from the white block will hinder $\mathcal{A}_1$'s plan.

But if the agents were to share information about their goals, they could deduce that they would get in each other's way if they don't cooperate, and instead try to form a joint plan. This sharing of information is done explicitly, by informing the other agent of one's goals.

Let's see what such a conversation would look like. Figure 1.5 shows the initial state, with agent $\mathcal{A}_2$ informing $\mathcal{A}_1$ about its goal. Figure 1.6 shows the agents communicating to perform a joint plan. $\mathcal{A}_1$ asks $\mathcal{A}_2$ to hold the white block, while it arranges the other blocks. In the process, it achieves $\mathcal{A}_2$'s goal, and when $\mathcal{A}_2$ replaces the white block on the gray block at 1, $\mathcal{A}_1$'s goal is also achieved.

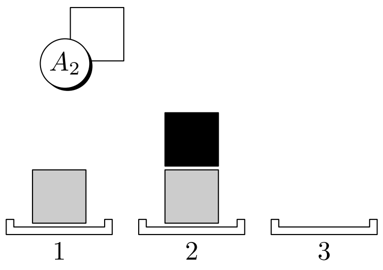

Figure 1.7: $\mathcal{A}_2$ holds the white block from 1 while $\mathcal{A}_1$ rearranged the other blocks

1.1.3 Conclusion

From the example interactions above we have seen that the communication between agents involves sending messages, either to *inform* about some fact, or to *request* that some action will be taken. The emphases on the words inform and request are not coincidental. In *Agent Communication Languages*, these kind of verbs are called *performatives*, and are used as labels in the messages, to indicate the type of communicative act the message is supposed to perform. Other types of performatives are possible, as we will see when we explore the theory behind speech acts in the following chapter.

1.2 Overview

This thesis is outlined as follows.

In the following chapter I will review state of the art agent communication technologies. To provide theoretical background for most of these technologies, the chapter begins with a brief overview of Speech-act theory. Then the technologies are reviewed and evaluated for their potential use when implementing communication in GOAL.

In Chapter 3, the semantics of communication constructs for the GOAL language are determined and specified. This is done by regarding communication from both a theoretical as well as a practical perspective.

Chapter 4 describes the implementation of these constructs. The choices made in determining syntax are explained. The chapter then goes on to describe the changes to GOAL's grammar and the GOAL interpreter.

The following chapter elaborates on how the communication in GOAL works and how it can be used. It explains the various programming constructs that are introduced and how these can be used to program multiple cooperating agents. This is demonstrated by building an example multi-agent system.

Chapter 6 describes mental models in more detail. The usefulness of mental models in programming with multiple agents is explained and demonstrated by extending the example multi-agent system from Chapter 5 with the use of mental models.

In Chapter 7 the issues when distributing agents across different machines are discussed. The GOAL interpreter is extended to allow agents to communicate across hosts. The focus here is to limit the impact this has on the burden of the programmer or the user of the GOAL platform.

Finally, the work done in this thesis is evaluated in the concluding chapter.

Chapter 2

Related Work

2.1 Introduction

In the field of multi-agent systems, many agent communication frameworks exist to support agent programmers in developing multi-agent systems in which agents communicate with each other.

In this chapter we will review several of these frameworks and explore how they relate to our objective of implementing communication for GOAL. Because communication in most of these frameworks is based on Speech-act theory, a brief review of this theory is given in Section 2.2. Section 2.3 reviews state of the art agent technologies. At the end of the chapter the technologies and their usefulnesses for our purpose are evaluated.

2.2 Speech-act Theory

Speech-act theory is based on the idea that speech is an act, and that certain instances of speech constitute additional acts being performed: an agent performs a speech act to change the mental states of other agents. Thus, speech acts are similar to "physical" actions that change the state of the world, except that they operate on mental states. The theory was introduced by philosophers Austin and Searle, but has enjoyed much attention from the computational field. Searle identified [15] a category of verbs whose "[...] utterance constitutes the performance of the act named by the performative expression in the sentence". E.g. with the sentence "I inform you that it is raining", the utterance of the sentence is called the *illocutionary act*. The *locutionary force* is that which the speaker wishes to achieve by performing the illocutionary act. In this case, informing the hearer that it's raining. The type of locutionary force is called the *illocutionary force*, which is in this case *inform*. Though precise definitions vary, these verbs are also called *performatives*.

2.2.1 Performatives

There are other performative verbs, like *request*, *order*, *promise* and *declare*. These are all performative verbs, because by speaking them the act they name is performed. By saying, "I request you close the door", one has *made the request*. By saying, "I promise you I'll come and visit", the promise *has been made*, regardless if it was a sincere promise. Characteristically, 'hereby' can be added to the sentence without altering it's meaning: "I hereby declare this mall as opened".

Not all verbs can be used performatively. Saying "I hereby wash the dishes" does not, unfortunately, make it so that I have washed the dishes.

This last sentence gives an example of the case that some verbs can only be used *performatively* if the speaker of the sentence is generally recognized, (i.e. has the authority) to perform the locutionary act. The situation and position of the speaker can influence the illocutionary force. Saying "The meeting is adjourned" will only adjourn the meeting if said by the chairman. If said by some normal meeting attendant, it might be interpreted as an 'inform' type of performative, instead of an 'adjourn'.

This distinction between types of performatives was discussed by Searle. He distinguished several classes of communicative acts:

- **Assertives**. By asserting a speaker commits itself to the truth of the asserted.
- **Commissives**. Through a commissive a speaker indicates that it is committed to perform the action mentioned.

- **Declarations**. Declarations make the content of the declaration true in the world (e.g. war being declared, two people being married).
- **Directives**. These are attempts to get the hearer to perform some action.
- **Expressives** indicate the speaker's emotional attitude toward some state of affairs.

That which distinguishes these classes are the conditions under which they can be successfully performed. These conditions apply to the mental states of the speaker and hearer.

2.3 Existing Agent Communication Technologies

Coming from Distributed Computing, where communication was already implemented by schemes as CORBA, RMI, RPC etc. ACLs are similar but provide more because they handle propositions, rules and actions as opposed to just objects without any semantics. And, an ACL message expresses a state rather than a procedure or method. Also, an ACL doesn't suppose a particular underlying transport mechanism or (usually) a knowledge representation language.

2.3.1 KQML

Starting in 1990, the Knowledge Sharing Effort (KSE) developed techniques and methodologies for reusing and sharing knowledge. To standardize the representation of knowledge, the KSE proposed the Knowledge Interchange Format. KIF was meant as an *interlingua*, a common language to represent the contents of a knowledge-base, which supports the translation to and from different native content languages [6][12]. This proposal encountered criticism [8], because it tried to standardize too much, which made it useless for most applications.

Sharing knowledge amongst autonomous entities implies / requires communication. To this end the KSE group introduced the Knowledge Query and Manipulation Language or 'KQML', a communication language and protocol for exchanging knowledge. It is meant to be a message-handling protocol and a message format to support run-time knowledge sharing among agents.

KQML specifies the syntax and semantics of messages. Figure 2.1 shows an example of a message in KQML.

KQML Listing

```
(ask-one
        :sender     joe
        :content    (PRICE IBM ?price)
        :receiver   stock-server
        :reply-with ibm-stock
        :language   LPROLOG
        :ontology   NYSE-TICKS
)
```

Figure 2.1: Example of a KQML message; agent joe asks the stock-server to tell it the price of the IBM stock

The syntax of a KQML message is a LISP-style balanced parenthesis, featuring a performative label and several key/value pairs. KQML provides an extensible set of *performatives*, which identify the *illocutionary force* of the message content. In this example, `ask-one` is the performative. The `content` field contains the actual knowledge in the native knowledge representation language. The contents of this field are independent from KQML and vice versa, with the exception that the `language` field should indicate which language is used there. This property allows KQML to be used in many situations, regardless of knowledge representation (KIF, SQL, Prolog, XML, . . .). The fact that the message itself can be represented in any encoding allows it to be transferred across many transportation media (TCP, SMTP, IIOP, . . .). This makes KQML a widely applicable ACL.

The set of performatives given by KQML is neither minimal nor fixed, instead it is meant to be extended by anyone when the need arises. It is up to the community implementing an agent system to determine which performatives from the basic set are to be used, and to implement additional performatives if necessary.

Communication Facilitators

Aside from the “normal” performatives identifying speech-acts, KQML introduces a small set of performatives which are used to describe meta-data regarding the information requirements and capabilities. Also, a special class of agents is introduced, called the *communication facilitators*. These special agents perform various communication services, like facilitating capability search, yellow pages, message routing, etc. The motivation for the introduction of these agents is that agents from any source should be able to join a network and be able to find their way in the social environment of agents, providing / advertising its capabilities and finding out about those of other agents.

Conclusion

KQML specifies a communication language, its syntax and semantics. The KQML specification lists 43 reserved performatives, with their meanings. It also defines exactly how new performatives should be defined. The set of performatives is so extensive and complex that agent builders will be inclined to define their own performatives. This de-standardizes the language, which makes it harder to use in a multi-vendor system.

Furthermore, the specification states that the performative definitions make reference to either or both the agent’s belief- or goal bases. However, the specification also states that the content of the messages is *inaccessible* to the protocol. This complicates verification of the agent’s program.

2.3.2 FIPA and FIPA ACL

Having a common message format enabled agent developers to have agents interact and exchange information regardless of their native knowledge representations. However, it does not facilitate the *interoperability* of agents, i.e. the ability of an agent society to operate with heterogeneous agents. Even with a common message format, agent designers of different agent systems or other technologies have no fixed specification of how to create or interpret those messages.

FIPA To address this issue, the Foundation for Intelligent Physical Agents (FIPA) was established. Its goal is to promote agent based systems and -technologies by developing specifications and interaction protocols, especially in the agent communication field.

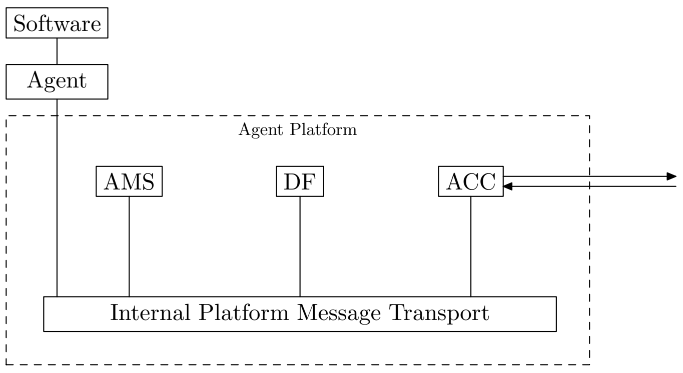

Figure 2.2: FIPA reference model of an Agent Platform

The first specification published by FIPA described the basic reference model of an agent platform, as shown in Figure 2.2, along with some special agents which are necessary for platform management. An Agent Platform (AP) is the physical infrastructure in which agents can be deployed. An AP does not bound the domain of the agent system, but is rather linked to a degree of locality. Agents from different APs may still interoperate with each other, and it is the frameworks responsibility to facilitate this in a transparent fashion, i.e. agents need not know whether some other agent resides on the same AP or not.

- The **Agent Management System** (AMS) is a mandatory component of every AP. It contains a directory of Agent identifiers which contain transport addresses for agents registered with the AP. The AMS offers white page services to other agents.
- **Directory Facilitator**s (DF) are optional agents that offer yellow page services to other agents. There may be more than one DF per AP, and they may be federated.
- The **Agent Communication Channel** (ACC) is the default communication method for agents on different APs.

FIPA ACL Agents communicate with each other by passing FIPA Agent Communication Language (ACL) messages. These messages consist of several message parameters. There are parameters for specifying the type of communicative act (performative), the participants in the conversation, the content of the message and the control of the conversation.

The transport mechanism used to transport the messages is not set by the FIPA standard, but it does give precise syntax description for ACL message encodings based on XML, text strings and several other schemes, allowing for any implementing transport mechanism.

FIPA ACL is inspired by and very similar to KQML, but with the important difference that FIPA ACL provides a formal semantics for the language, something that is seen as a shortcoming of KQML.

The semantics of FIPA ACL messages are formalized in the specification in terms of a communicative act's *feasibility pre-condition* (FP), and its *rational effect* (RE). Let's illustrate this with an example `inform`s semantic model.

$< s, inform(r, \varphi) >$
FP: $B_s\varphi \land \neg B_s(Bif_r\varphi \lor Uif_r\varphi)$
RE: $B_r\varphi$

Figure 2.3: FIPA `inform` semantic model

Here, s is the sending agent, r is the receiving agent, and the first line says that s informs r that φ holds. $Bif_r\varphi$ means that r 'knows' about the truth value of φ and $Uif_r\varphi$ means that r is 'uncertain' about the truth value of φ, i.e. it has no knowledge of the truth value of φ. This precondition that the receiver may not be uncertain about a fact in order to inform it about that fact is counter-intuitive to say the least. The reason that FIPA put this here is to ensure mutual exclusiveness of the feasibility preconditions among the communicative acts, when more than one communicative act might deliver the same rational effect. When a sender would believe that the receiver is uncertain about φ, it would send a `confirm` type communicative act, whose FP is $B_s\varphi \land B_sU_r\varphi$.

The feasibility precondition consists of two parts, the first being $B_s\varphi$, stating that the sender believes that the proposition it informs the receiver about holds. This is called the *sincerity property*, and relates to the *Gricean maxim of Quality* [9]. The second part of the FP, $\neg B_s(Bif_r\varphi \lor Uif_r\varphi)$, represents the *Gricean maxim of Quantity*, i.e. the sender does not try to inform the receiver about a fact that the receiver already knows about. In fact, it states that the sender should not adopt the intention to inform the receiver about φ even if it thinks r is only uncertain about it. In that case, it should perform another communicative act, like `confirm` or `disconfirm`.

The Rational Effect represents the *illocutionary force* of the message. It is the reason for sending of messages by the sender. If some agent s `inform`s another agent r that a door is closed, then it performs this communicative act because it wants the associated action be done (the receiver being informed about the state of the door). Depending on r's trust in s, r may adopt the belief that the door is closed.

The receiver may, upon "hearing" this message, conclude that the sender believed the proposition at the time of sending, and also that the sender wishes the receiver to believe that proposition. It is not, by this specification, *required* to adopt the belief that φ holds.

Communicative acts The FIPA Communicative Act Library Specification (Specification 00037) lists all communicative acts (CAs). Some of them have real performative meaning, like *inform*, *request*, *agree*, while others exist to accommodate conversations, negotiations and messaging (*propagate, subscribe, propose, reject proposal*). The specification lists 22 performatives, of which roughly half can be considered performatives in the speech-act sense, while the rest are either CAs to control the conversation or for relaying messages (*Proxy, Propagate, Not Understood*), or are variants of other CAs (*Inform If, Inform Ref, Request Whenever*).

Interaction Protocols Besides specifying the abstract agent architecture and the ACL syntax and semantics, FIPA has also specified a number of Interaction Protocols (IP). These IPs are pre-agreed message exchange protocols for using ACL messages to have complex conversations. Example IPs are the Contract-Net protocol and the well-known English- and Dutch Auctions. When an agent wishes to initiate a specific type of interaction (e.g. requesting an action), it can do so by setting the `:protocol` field of the ACL message to a protocol token identifying the IP used, and assigning a globally unique conversation-id to the interaction by setting the message's `conversation-id` field. Any receiving participant may respond according to the protocol, or reply that the message was `not-understood`.

For the case of doing a request to open a door, let's look at an example ACL message:

```
FIPA ACL
(request
        :sender            agent-1
        :receiver          agent-2
        :protocol          fipa-request
        :conversation-id   3228kjhIHIHI343
        :content           "open(door)."
        :language          PROLOG
)
```

Figure 2.4: An ACL message initiating a `request` interaction

In response to this request, `agent-2` may `agree` to or `refuse` the request. If the request has been agreed, `agent-2` should respond with either:

- a `failure` message, indicating that it has failed to execute the requested action, or
- an `inform-done` or `inform-result`, which are types of `inform` messages.

At any time during the interaction, the initiating agent may cancel the interaction by sending a `cancel` message. At any time during the interaction, any participating agent may reply to a message with a `not-understood` message, after which the interaction is terminated and any actions related to the IP may be considered not to have had effect.

2.3.3 JADE

The FIPA organization laid the foundation for agent frameworks that would allow agent developers to easily and consistently design agent systems. The Java Agent Development Environment (JADE) is a Java based software framework that enables agent programmers to develop agent applications in compliance with the FIPA specifications for interoperable intelligent multi-agent systems. It provides agent designers a set of tools, agents and systems for agent platform management, as well as a comprehensive API which developers can use to implement their agent logic in the framework.

Framework architecture

To conform to the FIPA specifications, JADE comes with a number of special agents that perform the tasks described in Section 2.3.2, like an AMS, DF, and the ACC. All these agents are automatically started when the agent platform is started. The agent platform can be distributed across multiple hosts. Each host runs one Java Virtual Machine, containing one Java thread per agent. An *Agent Container* is responsible for one or more of these agents, managing their life cycles and dealing with all the communication.

Communication

Communication between agents is done by constructing an *ACLMessage* object and sending it to the recipient. The way this sending is done depends on where sender and receiver are in the AP with respect to each other, but is completely transparent to the agents themselves. Several cases may be distinguished:

- Agents in same container: The ACL message object is not 'sent', but just *.clone()*d.
- Agents in same AP, different container: ACL message object is serialized and deserialized by RMI.
- Agents in different AP: The ACLMessage object is translated into a character string and then a remote method invocation on the remote platform's ACC is performed using IIOP as middleware protocol. On the receiver side, the string is parsed back into an ACLMessage object, which is then further relayed to the correct agent using the above method(s).

The ACLMessages are compliant with the FIPA ACL Message Structure Specification, and the API provides constants for selecting any of the 22 FIPA ACL Communicative Acts.

Message delivery ACLMessages are delivered at the receiving agent by placing it at the end of the receiving agent's message queue. It is up to the behaviour(s) of that agent to do either a non-blocking `receive`, which will return the ACLMessage at the head of the message queue, if any, or do a blocking `blockingReceive`, which will block the behaviour until a message is available in the queue. Both receive methods can take a `MessageTemplate`, which works as a filter. Only ACLMessages that fit the template will be returned by the methods. The template can match against any ACLMessage field. This way, agents can have specific behaviours that handle a certain protocol, for instance, or allow for conversation tracking.

Agent Execution Model

As mentioned earlier, JADE agents run in a single Java thread. This does not, however, restrict JADE agents to single-threaded behaviour, due to its flexible execution model.

JADE agents are defined in terms of *Behaviours*, which are abstract Java classes that are part of the JADE API. The Behaviour class has two methods, `action()` which defines the task to be performed, and `done()` which determines if the task is completed or not. By extending the Behaviour class and overriding these methods, one can implement behaviours for the agent. While all agents run concurrently in their Java thread, the behaviours of each agent are selected by the JADE agent by taking a behaviour from the ready queue and invoking its `action()` method. The behaviour is responsible for terminating its execution at some time, and if at that time `done()` returns false it is rescheduled for execution again. JADE agents schedule their behaviours within a single Java thread, using round-robin, non preemptive scheduling.

JADE comes with special subclasses of Behaviour for common tasks such as sending and receiving messages, and for complex behaviours composed of simpler ones.

Conclusion

JADE has enjoyed much attention and interest from the world of multi-agent systems, though not for its orientation towards FIPA compliance or extensive set of bundled protocols. The popularity of JADE is mainly due to it being open source, available and actively maintained and developed by a large community. It *has* risen to be the de-facto standard agent framework, but with some practical complications. For one, JADE has adopted and implemented all FIPA performatives into the ACLMessages, with the intention that they would be used to indicate *specific* speech acts, which would help interoperability with other agent systems. However, there are so many performatives that have lost most of their intuitive meaning or have an unclear meaning, that many agent designers resort to only using `inform`s, and putting their own meta-content constructs in the `:content` section. This issue is more thoroughly discussed in Section 2.3.5.

Another issue with JADE's implementation of communicative acts is that it claims to implement the semantics of the CAs as specified by FIPA. However, these specifications refer to the mental state of an agent, and JADE has no way of testing that mental state. On the contrary, JADE as a framework deliberately separates itself from the inner workings of the agents, leaving that domain to the agent programmer. To illustrate this, let's again take a look at Figure 2.3. The FIPA ACL specifications state that for an agent to perform an *inform* communicative act, the FP must hold. In this case, the FP refers to the beliefs of both sending and receiving agent. Since JADE cannot test these conditions, it cannot enforce FIPA ACL-compliant application of communicative acts.

2.3.4 Jadex

Jadex[2] is an agent framework which provides a reasoning engine based on the Belief-Desire-Intention (BDI) model for describing behaviour. Jadex uses JADE as middleware framework, and can thus be seen as a reasoning engine-extension of the JADE platform. As such, from our point of view it does not differ from JADE in terms of communication infrastructure and will therefore not be considered or evaluated separately in this thesis.

2.3.5 Issues

When considering the above ACLs and middlewares, we can observe that while KQML provides a standardized language for agent communication, its focus on the extensible set of performatives has actually inadvertently reduced its acceptance and use. Also, despite the proposals and designs of *facilitating agents*, no widely accepted or supported platform has been implemented providing these facilities.

To improve interoperability between agent systems FIPA standardized the set of performatives, and provides a sort of formal semantics of these communicative acts. This extensive standardization effort led to several implementations like FIPA-OS and others, the most used of which is JADE. JADE provides all the platform components that FIPA specifies and a framework for agent development.

As a middleware for GOAL, JADE could provide a means of agent communication and its platform can provide the directory facilities and the agent management functions. There are some features of JADE that when not used, do not cost any resources. This fits in JADE's design philosophy of 'pay as you go', meaning that you only pay runtime resources for the features that you use. However, since JADE is based on FIPA and FIPA ACL, it inherits their problems as well (see Section 2.3.2).

2.3.6 3APL

3APL is a language for programming cognitive agents with declarative goals. A 3APL agent has *beliefs* and *goals* as mental attitudes and a basic set of *actions* that it can execute when certain action-specific pre-conditions become satisfied. A 3APL agent can be programmed by defining the agent's beliefs, basic actions, goals and practical reasoning rules.

Communication

Originally, 3APL was designed as a single-agent APL. Two proposals have been made on extending 3APL with communication. The first proposal [11] distinguishes two types of message exchange. The first type is the *information exchange*, while the other deals with *making a request*. For each type, a pair of communication primitives is introduced. For information exchange they are, `tell` and `ask`, for making a request they are `req` and `offer`. It is important to note that these communicative primitives are *synchronous* communicative actions, which means that actual communication only occurs when two agents address each other. That is, communication has taken place when one agent makes a request (for information or action) and the other synchronously provides it with an answer.

The approach discussed in [11] does not try to find computational equivalents for speech acts, nor do they integrate conversation policies in the semantics of the communicative primitives. Rather, they focus on the requirement that the *receiver* of a message should be able to derive an answer to a question from that message. In the case of information exchange, this is done using *deduction*, and in the case of requesting using *abduction*.

The reason for this focus on the receiver is that an inherit problem with defining semantics for communication is the impossibility to predict the effect of a communicative act, without sacrificing autonomy. Even FIPA only states that the receiver of an `inform` message is "entitled" to believe its contents. The only assumptions the sender can really make according to the semantics given by FIPA for instance, is that the receiver has received the message and believes that the sender wishes to inform the receiver about the contents. More formally; $B_sB_r(I_sB_r\varphi \wedge B_s\varphi)$. It is exactly this issue that complicates the verifiability of communicating agent systems.

Criticism on this synchronous view is given in [4] on the grounds that we cannot pair all performatives that should be synchronized. Some performatives can have several different performative acts as response (`agree`, `refuse`) depending on the mental state of the receiver, while some performatives do not require or expect a response at all (one-way `inform`).

Instead, an approach is proposed in [4] that is based on asynchronous communication and supports modeling of FIPA-ACL performatives separately from the sending and receiving of the messages. 3APL agents send and receive messages to each other through the explicit 3APL actions `Send` and `Receive`. The actions are FIPA compliant in that they incorporate identifiers for the message, the sender, the receiver and a performative label indicating the type of communicative act. These actions are part of the 3APL language, while the content of the messages can consist of beliefs, basic actions, or goals. Messages are synchronously transported between the sender and receiver, but only by taking the message and putting it in the receiver's *message base*, until the receiver does a `Receive` action. The message base can be seen as a message buffer, where incoming messages await reception by the receiver. They exist as predicates in the form `received`$(i,\alpha,\beta,\rho,\varphi)$. At the sender's side, upon sending a message, a predicate `sent`$(i,\alpha,\beta,\rho,\varphi)$ is asserted in its message base. This 'delivery-on-demand' reintroduces asynchrony on the agent level. In this notation, i is a message identifier, α

identifies the sender of the message, β identifies the receiver, ρ is a keyword identifying the type of communicative act (i.e. performative), and φ is the message content.

The messages are handled at the receiving agent by means of practical reasoning rules, that have a logical formula that is tested against its message base in the rule's head. This way, rules can be defined that handle common protocol messages, like a *request* which should be answered with an *agree* if the requested action can be performed or a *refuse* if not. A programmer can change this behaviour if necessary by changing the practical reasoning rule, and can define message handling PR-rules for any situation.

Example Let's explore an example to illustrate what happens when agent *a* informs agent *b* about some fact φ. Figure 2.5 lists this conversation. Let $\mathcal{M}_i$, Σ_i, Γ_i be an agent *i*'s message base, belief base, and goal base, respectively.

Sender	Receiver
$\mathcal{M} = \emptyset, \Sigma = \langle\varphi\rangle, \Gamma = \langle s\rangle$	$\mathcal{M} = \emptyset, \Sigma = \emptyset, \Gamma = \emptyset$
$\mathcal{M} = \langle sent(1,a,b,inform,\varphi)\rangle, \Sigma = \langle\varphi\rangle, \Gamma = \emptyset$	$\mathcal{M} = \langle received(1,a,b,inform,\varphi)\rangle, \Sigma = \emptyset, \Gamma = \emptyset,$
$\mathcal{M} = \langle sent(1,a,b,inform,\varphi)\rangle,$ $\Sigma = \langle\varphi\rangle, \Gamma = \emptyset$	$\mathcal{M} = \langle received(1,a,b,inform,\varphi)\rangle,$ $\Sigma = \langle\Sigma_a\langle\varphi\rangle\rangle, \Gamma = \emptyset$

Figure 2.5: Example conversation, with *a* informing *b* about fact φ

2.3.7 2APL

Whereas 3APL is mainly a research project, 2APL aims more at supplying a usable platform on which agents programmed in the 2APL programming language can be executed and developed. The platform is written in Java and relies on the JADE middleware for agent management, communication and several tools that come with JADE, like the Remote Monitoring Agent, the message sniffer and the introspector. Agents can run on different hosts and can address each other conform the JADE addressing scheme.

2APL programs

2APL programs consist of several sections:

- Beliefs and goals
- Basic actions
- Plans
- Practical Reasoning Rules

Beliefs and goals are expressed as Prolog facts. In the case of goals, the facts reflect a desired state. Basic Actions specify the capabilities of the agent, and can act on the agent's belief base, the external environment, or can test conditions on the belief base or goal base. Also, Basic Actions can update the agent's goal base, by adopting or dropping a goal.

These Basic Actions are combined into *plans*. Plans can have an activation condition on the belief base, and they can be partially or wholly atomically executed. The plans can be dynamically generated through *practical reasoning rules*, which can fire upon events such as an incoming message from other agents, events from the external environment, or certain actions.

All this allows for complex plan generation and event handling.

Communication

A Communication Action is a type of Basic Action through which an agent can send a message to another agent. Such a send action has the following format:

```
2APL
Send(Receiver, Performative, Language, Ontology, Content)
```

Figure 2.6: 2APL Send action

Here, `Receiver` is a JADE-style agent address. It can be a local name for an agent that resides in the same agent container or a full name of the form `localname@host:port/JADE`. `Performative` specifies the speech act performed by this action, which can be any of the FIPA-ACL performatives. The following parameters `Language` and `Ontology` can be used to provide meta information on the content. These parameters are optional, because often agents will assume a certain language and ontology, so they are simply omitted. The final parameter `Content` is a representation of the content of the message.

Looking at the format of the send action in 3APL and 2APL we see some differences, which are listed below. The main reason for these differences is that 3APL is a goal directed APL that regarding communication adheres to the FIPA specifications which state that a message should contain `performative`, `sender`, `receiver` and `content` parameters, while 2APL is a more pragmatic implementation of an APL platform.

1. **2APL syntax does not include a message identifier**. The purpose of a message identifier is to distinguish one message (in the message base) from a potentially identical other message. Because 2APL is built on JADE, incoming messages will be *queued* at the receiving agent, not *pooled* in a message base as in 3APL.

2. **2APL syntax does not include the sender's identifier**. In 2APL, message events are used as triggers for message handling procedural rules. These events have the form `message(Sender, Performative, Language, Ontology, Content)`.

3. **3APL syntax does not mention language or ontology**, because 3APL specifies that the content can be only beliefs, basic actions, or goals, so a choice of language or ontology would be unnecessary.

2.3.8 JASON

Jason is an implementation and extension of the programming language AgentSpeak. It is designed with the idea in mind that Jason agents should be able to communicate and cooperate on the knowledge level, meaning that they can communicate in terms of their beliefs, goals, and plans.

In Jason, mental states consist of *beliefs*, facts which they perceive or deduce from other facts, events in the environment for which they may have *event handlers*, and *intentions*, which are goals that are currently being pursued. This 'BDI' model has been eagerly used in the artificial agent field, because modeling agents in this way provides the agent programmer with an intuitive link with how humans are known to think and act. Also, because the activity of selecting plans for action is separated from the execution of the plans, BDI-agents have the ability to balance between different intentions, so they do not over-commit themselves to plans that have lost their use, and can as such be more effective.

Jason agents are programmed in AgentSpeak, and consist of two parts. The first part contains the agent's initial beliefs and goals. In the second part the agent's *plans* are defined. These plans are 'recipes' for how to act upon certain events in certain conditions. Plans have the following format:

Jason plan
```
<triggering effect> : <context> <- <body>
```

The `<triggering effect>` specifies for which events this plan is relevant. A plan is only applicable for execution if the plan's `<context>` is entailed by the agent's beliefs. The `body` lists the *basic actions* that are to be performed when this plan is executed.

There is also an optional construct to be used in the belief base, an annotation which captures the *source* of the information. It is a list of terms enclosed in square brackets. Two sources are predefined, `percept` and `self`, but sources could be other agents, through communication.

Communication

A Jason agent can communicate with other agents by sending messages using the *internal action* `.send`, with the following format:

Jason send action
```
.send(receiver, illocutionary_force, propositional_content)
```

`receiver` is a label for an agent as defined in the multi-agent system. `illocutionary_force` represents the type of performative for this message. The illocutionary forces available in Jason are: `tell`, `untell`,`achieve`, `unachieve`, `tellHow`, `untellHow`, `askIf`, `askOne`, `askAll` and `askHow`.

An interesting property of Jason agents is their ability to not only exchange beliefs and goals, but also plans. This way agents can tell each other *how* to act upon certain triggering events. The `*How` performatives are used for this kind of communication. `achieve` and `unachieve` are used to request that the receiver achieve a certain state in the world, or stops doing that, respectively. `tell` and `untell` are used to indicate a sender's intention to have the receiver believe the propositional content to be true, or to stop believing the propositional content to be true, respectively.

Messages are sent and received asynchronously. When sent, the message is put in the receiver's mailbox. At the beginning of the receiver's reasoning cycle, one message is selected from the mailbox to be processed. This selection of messages is subject to a user-definable function which determines if a message is *socially acceptable*. For example, an agent may want to reject a `tell` message from an agent that it does not trust. If a message passes this function it is then treated as an event, possibly triggering plans in the agent's reasoning process. Annotations specifying the source can be used here to valuate a message in the social acceptability function, or as part of the event trigger in the plan rule. This way, an agent can handle information it perceived itself differently from information it received from other agents.

Conclusion

While the illocutionary forces of the communication in Jason are inspired by KQML, the set of performatives is relatively less extensive and complex compared to that of the KQML specification. There are 9 performatives, and the semantic rules for interpreting received messages with each of the performatives are given in [18]. A specific focus of Jason is to be able to communicate in terms of beliefs, goals and plans.

2.4 Conclusion

As noted in the Introduction, when considering a communication infrastructure for GOAL, the following criteria are observed:

1. The communication constructs should have a well-founded theory.
2. The distinction between beliefs and goals should persist in the communication.
3. Programming agents using the communication constructs should be pragmatic, and pose a small burden on the programmer.

Several agent platforms, frameworks and agent programming- and communication languages were reviewed. Some of these have a strong link with Speech-act theory, and try to map the theory with all its performatives onto an agent communication language. Often, additional performatives are introduced to facilitate meta-communication, such as services discovery, message routing etc.

The technologies we have reviewed in this section give us a technical framework for communication, but most have issues concerning the formal verifiability of the programs when used in a multi-agent environment. Going from KQML to FIPA (ACL), efforts have been made to provide formal semantics of the communication language, but these semantics refer to the beliefs and goals of the sending agent, which is not realistic since there is no way to inspect another agent's mental state directly.

JADE provides a framework and platform for developing FIPA compliant multi-agent systems. The tools and the technical framework are powerful, and are used in several other agent platforms. But since the communication is based on the principles of FIPA, it also inherits its issues with formal verification.

All these technologies suffer from lack of formal semantics, and a performative set that is ambiguous and vague which leads to misinterpretation by agent designers, thus defeating the interoperability those technologies aim to facilitate.

These problems were acknowledged within the field, and due to an increasing demand for verifiable multi-agent systems, attempts have been made to address them.

3APL and 2APL take a different approach at the implementation of the communication process, by modeling the FIPA-ACL performatives separate from the sending and receiving of messages. By asynchronously sending messages via a message base, the semantics of the communication primitives allow for better formal verification. Jason tries to reduce the performative set, defining performatives for sharing information (beliefs), goals, plans and asking for information.

These latter APLs have better theoretic foundations, but still do not formally specify what the receiver of a message should actually do with it. Overall these technologies can give us a technical framework for communication, but fail to satisfy our first criterium of having a well founded theory.

Also, with the exception of 3APL, no ACL makes any distinction between beliefs and goals in the communication. Propositional content is just packed in a message, and the programmer can apply an appropriate label (performative) on it. But these labels bear no relation to the semantics of the ACL itself. Our second criterium is therefore not satisfied.

Often, because of too vast, vague and ambiguous performative sets, agent programmers tend to improvise when using performatives, hampering interoperability and complicating the agent programs. This is clearly not a pragmatic way of programming agent communication. Handling these situations places a burden on the programmer, which violates criterium 3.

None of the ACLs that we reviewed in this chapter give us a framework for communication for GOAL which has a formal semantics based on a well-founded theory and provides a pragmatic way for agent programmers to have agents communicate their beliefs and goals.

This makes these technologies unsuitable for direct application as a communication middleware for GOAL, as they do not meet the criteria that were listed in Chapter 1 and reiterated at the top of this section.

Chapter 3

Semantics

3.1 Introduction

So, what is communication, in the context of agents? Communication is the successful conveyance or sharing of information. When we communicate, we want the act of communication to have an effect on the audience. For example, suppose I (the speaker, s) tell you (the hearer, h) "It's raining." (φ). From my point of view as speaker, I would like to believe that the utterance of the sentence has an effect on the hearer, namely that the hearer believes that which was asserted.

In this chapter I discuss why the above is not a trivial matter, and how this causes a fundamental problem with many existing agent communication languages. Instead of a communication semantics based on Speech-act theory, I introduce an alternative approach to agent communication, in which the autonomy of the agents is respected, and is inspired by linguistics.

First, the objections to the conventional speech-act inspired approach are discussed.

3.2 The effect of communicating

The effect of the act of uttering "It's raining" might be either (or any, or all) of these effects;

1. You come to believe that it is raining: $B_h\varphi$
2. You come to believe that I believe that it is raining: $B_hB_s\varphi$
3. You come to believe that I had the intention to make you believe that it is raining: $B_hI_sB_h\varphi$
4. Nothing. Or, more politically correct, 'anything'.

The last effect in the list suggests that something is going on here. Indeed, we cannot strictly assume that the first effect will actually happen; I may not have convinced you. So effect 1 is too presumptuous.

Effect 2 is somewhat less gullible. It doesn't simply adopt the belief in whatever it is told, but instead assumes that I believe what I say. This is not always a safe assumption. For example, when I am lying, I do not believe what I say, so believing that I believe what I say would still be overly credulous.

So let's take another step backwards to effect 3. This is an even safer statement, assuming only that one does things because one intends to do those things. In this case, 'things' refers to having you believe the truth of φ. But from my point of view, knowing that you believe that I intended to have you believe that it's raining is of little direct use, unless I trust that you do *something* with that first belief.

But that is exactly the issue with this kind of communication. Can a speaker assume anything about what a hearer does with this information? What actually happens when we utter a sentence? Of course, the act of speech brings about more than just the knowledge *that* something has been said. Exactly what it is that these speech acts do aside from the trivialities of being performed (uttered), heard, etc, is the core topic of Communication Theory.

These issues mentioned in the above paragraphs were noticed in natural language communication by philosopher Paul Grice [9], who studied the discrepancy between what is meant by an utterer and inferred by a hearer. Grice identified the *moods* of utterances as *conversational implicatures*, a term roughly defined as ‘things that a hearer can make out from the *way* something was said rather than *what* was said’. When John from Figure 3.1 utters “Mary, the salt is on the table.”, he does so in

```
‘The salt is on the table’ in three moods
John: ‘‘Mary, the salt is on the table.’’
John: ‘‘Mary, is the salt on the table?’’
John: ‘‘Mary, I’d like to see the salt on the table please!’’
```

Figure 3.1: John and Mary converse

an indicative mood. When he utters “Mary, I’d like to see the salt on the table please!”, it is in an imperative mood.

Moods were also a subject of research for Robert Harnish[10], who distinguished major from minor moods. Examples of moods in the English language are indicative, subjunctive, imperative, infinitive, participles, expressives. He categorizes major moods as being:

1. highly unrestricted in their productivity
2. central to communication
3. high in relative frequency of occurrence
4. common to most languages

He then identifies three major moods:

- declaratives
- imperatives
- interrogatives

Instead of ‘declarative’, we will use the term ‘indicative’ for the same mood. These moods correspond to the moods of the sentences in Figure 3.1. In general, sentences can be decomposed into

1. the part that represents the propositional content of the sentence. In these examples, that is “the salt is on the table”. Grice calls this moodless element the *sentence radical*.
2. the *mood operator*, which can be $\vdash$ for the indicative mood, ! for the imperative mood and ? for the interrogative mood.

Both elements are combined to form a notation $\star(R)$, where $\star$ is a mood operator and R is a sentence radical, for representing the meaning of sentences, and is explained as follows. Suppose U is an utterer, uttering in the direction of audience A.

1. U means $\vdash(\varphi)$ by uttering x if and only if U intends A to think U thinks that φ.
2. U means $!(\varphi)$ by uttering x if and only if U intends

   i. *A* to think that *U* intends to bring about φ
   ii. *A* to intend φ

3. *U* means ?(φ) by uttering *x* if and only if *U* intends *A* to think that *U* intends to know the truth of φ

This notation models utterances by isolating the sentence radical from the mood operator. Searle ([16], p47-50) also made this separation between what he calls the illocutionary force and the propositional content in his amendment of Grice's account on Meaning, for his own discussion of speech acts. Searle expanded Grice's definition of non-natural meaning to account for convention in communication. Searle also uses a similar notation for modeling speech acts.

This notation and Grice's analysis of communication and conversational implicatures give us a better basis for designing communication constructs for agents than pure Speech-act theory. These constructs are proposed in Section 3.6.

3.3 Mental vs. social agency

So far, the multi-agent architectures we have considered all have (formal) semantics that emphasize *mental agency*, the supposition that agents should be primarily understood in terms of their mental attitudes, such as beliefs, goals and intentions. This emphasis on these mental concepts in the formal semantics of agent communication languages was criticized by Singh ([17]), who argued that this could not work for agents that aim to be both autonomous and heterogeneous, because it assumes agents can read each other's minds. He argues that the semantics of receiving communicative acts refer to the mental state of the sender, which is not accessible or verifiable in most practical applications.

Singh voices two major objections to this view:

1. Referring to the mental state of the sender leads to reduced autonomy, both in design and execution of the agent programs.

2. Dialects and idiolects of the communication language form.

Instead, he suggests to approach agent communication and interoperation with *social agency*, which views agent communities as *societies*, in which agents play different *roles*. The rules for communication between agents in a society are specified by that society's *protocol*. A protocol specifies the commitments that an agent playing a certain role must adhere to. For example, in an online bookstore society, an agent playing the role of a seller, must accept a purchase of an item for the price that it offered and was accepted.

Because the protocols are expressed in terms of these commitments, any agent's compliance to the protocol can be tested by observing that agent's communicative acts. This means that agents can join societies and be accounted for their actions, without having to explain their internal workings. The society of agents will be the enforcer of the protocols. When, for example, an agent asserts some proposition, it is socially held to the truth of this assertion. If it seems that the proposition was falsely asserted, the offending agent can be regarded in the society as untrustworthy. Singh envisions that agent vendors will design protocols for specific applications and that those protocols will evolve as different agent developers contribute.

This suggested *social semantics* should facilitate design autonomy, because the requirements of agents act not on the implementation (e.g. by imposing mental state conditions per communicative act) but instead act on their behaviour. Execution autonomy, an agent's freedom to choose its actions, is also sustained by social semantics, because an agent is not restricted in its actions, as long as it obeys the rules of the society and the behaviour dictated by the protocols belonging to its roles. A platform for managing societies and roles would be required to have multiple agents interoperate and fulfill certain roles.

Our approach Our view on formal semantics of an agent communication language for GOAL shares the idea that it is not possible for one agent to directly inspect another agent's mental state. We suggest an approach that respects autonomy by not putting mental state conditions in the semantics of communicative acts.

3.4 Conditions on the sender

Speech-act theory banks on the notion that communication relies on that the speaker's utterance is a manifestation of its intention to *change the beliefs* of the hearer, and that the hearer's *recognition* of that intention from the utterance. The emphases lie on the speaker's intention on one hand, and the hearer's processing of the heard utterance on the other hand.

Sincerity A key concept in speech acts is sincerity. The sincerity of an utterance is the degree in which the perceived intention of the speaker matches it's actual intention. The inability to, as hearer, reliably inspect the speaker's mental state (and thus intentions), is inherit to communication. Or, seen the other way around, it is the reason for communication. For if we were able to inspect each others mental state, we could replace the whole concept of communication with that of telepathy. But that is not the case in the definition of agents, be it human or software.

Conversational maxims The intention of U is clear to A only if the utterance adheres to certain conditions, called *conversational maxims*. Grice categorized these maxims into the following four maxims:

1. **Maxim of Quality**. Do not say what you believe to be false, or for which you lack adequate evidence.
2. **Maxim of Quantity**. Be as informative as, but no more informative than, necessary.
3. **Maxim of Relation**. Be relevant to the topic.
4. **Maxim of Manner**. Avoid obscurity. Be concise.

These maxims can be seen as conversational principles that audiences use in order to construct an inferential bridge between what is meant and what is implied. If U utters "The salt.", H cannot reasonably be expected to deduce U's intention to have H put the salt on the table; the maxim of Quantity is violated. Similarly, stating that "The bus will arrive at 10:30 if whales are mammals." to inform H that the bus will arrive at 10:30 violates the maxim of Relation.

Perhaps the most important maxim is the maxim of Quality. Without observing this principle, conversation would be rather cumbersome. If *U* cannot be obliged to adhere to the maxim of Quality, then asking *U* what time the bus arrives will be useless, because nothing guarantees that *U* actually believes what he says. Note that we are not after the 'universal truth', only what someone believes to be true. When we ask someone for the arrival time of the bus, we expect that person to respect the maxim of Quality and respond *to the best of his knowledge*.

Maxims for agents These maxims are as relevant for agent communication as for human communication. When an agent *A* receives a message from sending agent *S*, it will only think that *S* believes the contents of that message by virtue of the assumption that *S* respects the conversational maxim of Quality. There is no direct way to ensure this, especially across different agent builders, apart from convention.

Assuming the maxim of Quality is respected, receiving agents may also assume that

- $receivedFrom(a, comm(b, \vdash(\varphi))) \rightarrow \Sigma_a \models \varphi$
- $receivedFrom(a, comm(b, !(\varphi))) \rightarrow \Sigma_a \not\models \varphi \wedge \varphi \in \Gamma_a$
- $receivedFrom(a, comm(b, ?(\varphi))) \rightarrow \Sigma_a \not\models \varphi$

Note that these assumptions do refer to the actual mental state of the sender, but since they are just assumptions no technical ability to verify them is required.

3.5 What we can do

Let's take a look at our example agents from Section 1.1. In order to resolve the conflict in their combined goals, they communicate their beliefs and goals. Three types of communicative acts are used in this communication:

1. indicatives. These are statements about one's own beliefs.
2. interrogatives. These are questions about a certain fact. They should not be considered as direct queries on another agent's belief base, but rather as requests to inform the speaker about a fact.
3. imperatives. These are statements about one's own goals. Again, these should not be considered as direct insertions of goals into another agent's goal base, but rather as an indication that the speaker has that goal.

Informally, we can describe the semantics of these communicative acts as follows. Suppose *A* is the speaker and *B* is the hearer of the communicative act.

Indicative *A* asserts φ. *B* hears this assertion, and can conclude that *A* (thinks it) knows φ.

Interrogatives *A* queries *B* about φ. *B* can conclude that *A* did not know, or was uncertain of, the truth of φ.

Imperatives *A* states that it has φ as element of its goal base. *B* can now conclude two facts: *A* has φ in its goal base, and *A* does not believe φ. If *A* would believe φ, it would no longer be a goal.

It is remarkable that none of these informal semantic definitions define what the hearer should do with the speaker (like replying to a query), except deducing facts about the speaker's mental state. This conforms to the view that two autonomous agents cannot look inside each other's mental state, let alone change it. What we *can* do is deduct an agent's mental state from the communicative acts it performs.

Figure 3.1 showed three sentences uttered by John in the direction of Mary. These sentences are in an indicative, interrogative, and imperative mood, respectively. They all act on the same fact: "the salt being on the table". Let's call that proposition φ. Apart from the order of words, the most distinctive feature indicating the mood of the sentence is the punctuation marks at the end. We see here the use of a period '.' for the indicative sentence, a question '?' mark for the query, and an exclamation mark '!' for the directive sentence. Using these punctuation symbols in a formalized communication language, we could communicate φ in the three different moods by annotating φ with either symbol. For the period symbol we make a slight exception. In fact, we adopt the symbols used by Pendlebury[13]; ':' for the indicative, '?' for interrogative and '!' for the imperative mood.

3.6 Syntax and semantics of the communication language

One of our criteria for a communication implementation in GOAL is that the communication language should have a formal semantic definition. In order to specify that we must first formally define the syntax.

$$msg ::= \, :\varphi \mid ?\varphi \mid !\varphi \tag{3.1}$$
$$comm ::= comm(agt, msg) \tag{3.2}$$
$$agt \in \textit{Agent names} \tag{3.3}$$

Figure 3.2: Syntax of the mood operators

We have replaced the symbol '$\vdash$' used by Grice to indicate the indicative mood by ':', as explained above. The symbols then match the semantics seen in Figure 3.1.

Here we attempt to formally describe the semantics of the communication primitives mentioned above. Informally, upon receiving (or hearing) a message, an agent updates its model of the sending agent's mental state. To formalize this definition, we must first give the definition of a mental state.

A differentiation is made between *basic mental states* and *complex mental states*

Definition 1 *Let $\Sigma \subseteq \mathcal{L}_0$ be a belief base, and $\Gamma \subseteq \mathcal{L}_0$ be a goal base. Then a* basic mental state *is defined as*

$$\mathcal{M}^B = \langle \Sigma, \Gamma \rangle$$

Definition 2 *A* complex mental state *is defined as*

$$\mathcal{M}^C = \langle \Sigma, \Gamma, m \rangle$$

where

$$m : \textit{Agent names} \rightarrow \mathcal{M}^{B}$$

is a function that maps agent names to (basic) mental states.

Definition 3 *An* agent *is defined as:*

$$A = \langle a, \mathcal{M}^{C}, \Pi \rangle$$

where a is a name.

Definition 4 *A* multi-agent system (mas) *is a set of agents:*

$$mas = \{A_0, \dots, A_n\}$$

with A_i an agent whose name is unique throughout the mas.

The structured operational semantics of the three communicative acts are given by

$$\frac{A = \langle b, \Sigma, \Gamma, m, \Pi \rangle \in mas, A_a \xrightarrow{com(b, \odot\varphi)} A'_a}{mas \rightarrow mas\backslash_{\{A\}} \cup \{A'\}}$$

where:

- $A'_a = \langle b, \Sigma, \Gamma, m', \Pi \rangle$, i.e. an agent with name b.
- $m(a) = \langle \Sigma_a, \Gamma_a \rangle$, where Σ_a and Γ_a are the belief- and goal base of a, respectively.

The mapping function m returns a mental state model of a given agent. It represents what an agent ‘thinks’ another agent’s beliefs and goals are. This information can of course be outdated and inconsistent with the modeled agent’s real beliefs and goals. The receiving agent can update its mental model upon reception of a communicative act in a way that is described below.

In the above operational semantics, m' is the updated mental state mapping of the receiving agent as a result of a communication from agent a: $com(b, \odot\varphi)$. Here, $\odot$ represents one of the three communication symbols: $:, !, ?$. If $m(a) = \langle \Sigma_a, \Gamma_a \rangle$, then for each of these symbols, the semantics is given by:

1. **: (indicative)** $m'(a) = \langle \Sigma_a \oplus \varphi, \Gamma_a \setminus \{\gamma \in \Gamma_a | \langle \Sigma_a \oplus \varphi \rangle \models \gamma\} \rangle$. If a asserts φ, then the receiver may assume that a believes φ, and thus also that it has no goal to achieve φ. The $\oplus$ operator represents the `insert` operation of the KRT. The model of the goal base is updated by removing any goals that are entailed by the new belief base. This is to maintain mental state consistency; an agent should not have a goal to achieve something that it already believes to be true.

2. **! (imperative)** $m'(a) = \langle \Sigma_a \ominus \varphi, \Gamma_a \cup \{\varphi\} \rangle$. If a indicates it wants the state φ being reached, then the receiver may assume that a does not believe that φ is the case, and also that φ is a goal of a. The $\ominus$ operator represents the KRT’s `delete` operation[1].

[1]The semantics of the $\ominus$ operator should be nuanced in that it does not delete facts that are not entailed by the belief base. I.e.: $\Sigma \ominus \varphi = \Sigma \; if \; \Sigma \not\models \varphi$.

3. **? (interrogative)** $m'(a) = \langle \Sigma_a \ominus \varphi, \Gamma_a \rangle^1$. If a asks about some statement φ, the receiver can assume that a does not know the truth value of φ, or was not certain about φ. If, up until now, the receiver thought that *a did* believe φ, it updates its mental model of a to reflect the new information.

This semantics does not refer to the *actual* mental state of the sender, nor does it define when a sender should send a message or what a receiver should do with the contents of a received message other than simply record it in its mental model of the sending agent. As is argued earlier in this document, it is infeasible to make formal statements on the receiver side based on the mental state of the sender.

The semantics does make some implicit assumptions with respect to the sending agent's behaviour. For example, for the indicative mood, it is assumed that if one does an assertion, it believes the contents of that assertion itself. This is not a trivial assumption, but is based on conversational implicatures that originate from philosophy, which was discussed and made explicit for the GOAL implementation in Section 3.4.

3.7 Querying mental models

In order to inspect the mental models, additional belief and goal operators are introduced to the GOAL language. These operators resemble the existing belief and goal operators for querying the agent's own mental state. The model of another agent's belief- or goal base is queried by specifying the agent's identifier as the first argument to the operator:

GOAL

```
bel(<agent identifier>, <belief query>)
goal(<agent identifier>, <goal query>)
```

The semantics of these operators are as follows:

Definition 5 *Semantics of the belief model query operator:*

$$\texttt{bel(}\mathrm{a}, \varphi\texttt{)} = true\ iff\ \Sigma_a \models \varphi$$

Definition 6 *Semantics of the goal model query operator:*

$$\texttt{goal(}\mathrm{a}, \varphi\texttt{)} = true\ iff\ \exists \gamma \in \Gamma_a : \gamma \models \varphi \wedge \Sigma_a \not\models \varphi$$

3.8 Example usage

An example program using the suggested approach for communication is shown in Figure 3.3. The program is a simplified version of a bomb-cleaning agent. Bomb-cleaning agents operate in an environment where bombs are located at specific locations. The environment is partially observable, in the sense that bombs can only be detected (sensed) in a certain finite radius around the agent. Agents have the goal to remove all bombs from the (finite) environment by picking them up and bringing them to the trashcan which is located at a known fixed location. Agents can hold only one bomb at a time.

GOAL

```
 1 knowledge {
 2   clean(X,Y) :- not bombAt(X,Y).
 3 }
 4
 5 beliefs {
 6   agent(b).
 7 }
 8
 9 program {
10   % Ask the other agent to help us, if we have our hands full
11   if bel(bombAt(X,Y), holding), a-goal(clean(X, Y)) then send(b, !(clean(X,Y))).
12
13   % Inform the other agent of bomb locations it doesn't seem to know about
14   if bel(bombAt(X,Y)), not(bel(b, bombAt(X,Y))) then send(b, :(bombAt(X,Y))).
15
16   % Inform the other agent that it's goal has been achieved
17   if goal(b, clean(X,Y)), bel(clean(X,Y)) then send(b, :(clean(X,Y))).
18
19   % If we know of no bomb location, query the other agent for one
20   if not bel(bombAt(_,_)) then send(b, ?(bombAt(X, Y))).
21
22   % If we have nothing better to do, help out the other agent
23   if not(goal(clean(_,_))), goal(b, clean(X,Y)) then adopt(clean(X,Y)).
24   % Or, adopt a goal using the other agent's beliefs
25   if not(goal(clean(_,_))), bel(b, bombAt(X,Y)), not(goal(b, clean(X,Y))) then adopt(clean(X,Y)).
26   % Or, adopt a goal using our own beliefs
27   if not(goal(clean(_,_))), bel(bombAt(X,Y)) then adopt(clean(X,Y)).
28 }
29
30 actionspec {
31
32   pickup {
33     pre{ not(holding), bombAt(X,Y), at(X,Y) }
34     post{ holding, not(bombAt(X,Y)) }
35   }
36 }
```

Figure 3.3: An example GOAL program

For brevity and simplicity, we assume there are only two agents active in the environment, with similar programs that differ only in the names of the agents. Figure 3.3 shows the program for agent '*a*'.

Program rules and actions to bring the bomb to the trashcan are omitted. The first three program rules handle outgoing communication. All three communicative acts are being used in those rules:

- If we want something done, send a request (line 11). The updated mental model in agent *b* is: $\mathcal{M}_a' = \langle \Sigma_a \ominus \texttt{clean(X,Y)}, \Gamma_a \oplus \texttt{clean(X,Y)} \rangle$

- If we know something we think the other agent does not know, send an inform (lines 14 and 17). The updated mental model in agent *b* is: $\mathcal{M}_a' = \langle \Sigma_a \oplus \texttt{clean(X,Y)} \rangle$

- If we want to have more information, query the other agent for bomb locations (line 20). The updated mental model in agent *b* is: $\mathcal{M}_a' = \langle \Sigma_a \ominus \texttt{clean(X,Y)} \rangle$

It should be noted that all sentences in the communications are grounded, and the `X`s and `Y`s above will have been substituted.

By communicating, these agents will have more knowledge about the environment, and perform more efficient. When an agent is moving while holding a bomb, and encounters another bomb, it can request the other agent to take up the task of removing it. Technically speaking, the agent only notifies the other agent that it has this goal, but when looking at the whole program (especially line 25), the effect is the same.

3.9 Self-referential communication

Section 3.7 shows the syntax and semantics of the belief- and goal operators for querying the *models* of the mental states of other agents. The operation looks like the conventional ‘local’ query of the agent’s own mental state, with the distinction that they operate on a *model* of a mental state rather than a *real* mental state.

But what exactly is the difference between those? Technically, they are very similar. The implementation of a mental state model is identical to that of a normal mental state. The difference lies in the addressing of a model. Since an agent can have mental state models of multiple agents, the querying of a mental state model involves retrieving the right model from the collection of models, i.e. an array indexing, more than querying a normal mental model.

Considering this, querying a normal mental state seems like a special case of querying a model. In fact, we could generalize the querying by viewing querying the own mental state as a case of querying the mental model of oneself. In other words, instead of querying (and maintaining) our own mental state, we query *a model* of our mental state, amongst the other models. This would eliminate the need for storing our own mental state separately. The querying could be done in the following manner where `self` is an alias for the agent’s own identifier:

GOAL

```
bel(self, bombAt)
```

But how about updating our mental state? Since we no longer maintain a conventional mental state, the update operations `adopt`, `drop`, `insert` and `delete` no longer apply. How do we, say, insert a belief in our belief base, or adopt a goal? Fortunately, the proposed constructs for communication already provide this functionality. By ‘sending’ an `inform` to ourselves, the communication handling mechanism will update our model of our own belief base. Likewise, sending a `request` will remove the associated fact from the belief base, and add it to the goal base. Removing a belief from the belief base can be done by sending a `query`.

This way, conventional belief- and goal bases are no longer needed. Whether this approach is desired depends on the burden it imposes on the agent programmer.

3.10 Conclusion

To investigate the integration of communication support into GOAL, we first determined which criteria such a solution would have to satisfy. Special attention has been given to the requirement of a well-founded theory with high-level semantics. Also, the solution should provide simple yet useful primitives for the agent design.

In response to the *mentalistic* view of the techniques discussed in the previous chapter, Singh proposed that since we cannot read each other's mind, we should instead focus on the social commitments agents make as a result of their communicative actions. Social commitments are common knowledge, since they can be tested by observing an agent's (communicative) actions. The problem here is that the distinction between beliefs and goals, something we need for GOAL, disappears.

Because a fundamental characteristic of communication is that you have no guarantee that your communicative act will have the intended effect on the hearer's mental state, our suggested approach for an ACL for GOAL is based on the idea that even if a speaker *cannot* assume anything about the mental state of a hearer as a result of a communicative act, at least the hearer *can* deduct the mental state of the speaker, and keep an updated model of that agent's mental state. The semantics of the mental models and querying them have been specified in this chapter.

A formal syntax and semantics are proposed as language constructs for 3 major communicative acts: informing, querying and requesting, represented syntactically by the symbols '`:`', '`?`' and '`!`', respectively.

The example presented in Section 3.8 looks promising. The communication primitives and mental model querying constructs are the pragmatic, clear tools for the programmer that we sought to determine.

The following chapter describes the implementation of the communication primitives and mental models based on these semantics.

Chapter 4

Implementation

4.1 Introduction

After having determined the requirements and the semantics, the changes to the GOAL interpreter had to be implemented. Because the present GOAL interpreter was oriented around reasoning only about the agent's own beliefs and goals, changes were necessary in the GOAL grammar to allow GOAL agent programmers to express conditions in terms of beliefs and goals of other agents, and to perform `send` actions. Also, the interpreter had to be changed to accommodate keeping a mental model of each known agent.

Summarizing, the following needed to be facilitated:

1. perform mental state queries on mental models of other agents, so some sort of model selection construct is necessary
2. perform a send action, which is a reserved GOAL action, which will send some message in a specific mood to a list of agents.

There are also several constraints that need to be observed in determining a syntax for the above-mentioned constructs.

1. Using the communication constructs should be *uncomplicated and intuitive*. Reading a GOAL program which uses them should be reasonably easy to understand without knowing it's precise semantics.
2. The grammar should be *backwards compatible*, meaning that any GOAL program which was written without communication should still work as-is in the new interpreter.
3. The constructs should *allow enough expressivity* to be useful in practice.

In this chapter I will elaborate on the implementation process and describe the manner in which the implementation was decided upon. First I will describe the changes made to the GOAL grammar. Then the process of changing the interpreter is discussed, and finally I will describe how communication was realized.

4.2 Extending the GOAL grammar

The GOAL grammar defines the syntax of GOAL agent programs. It is the nature and defining characteristic of GOAL programs that goals are defined declaratively, which makes it possible to program the reasoning of the agent in terms of the agent's beliefs and goals. A GOAL program rule therefore has the general format as shown in Figure 4.1.

GOAL
```
if <mental state condition> then <action>.
```

Figure 4.1: The format of a GOAL program rule

This `mental state condition` is an expression of beliefs and goals that the agent has. In GOAL there are four operators that can be used to query the mental state;

- bel(φ)
- goal(φ)
- a-goal(φ)
- goal-a(φ)

Here, φ is an expression in the agent's KR language. In this format, the mental state condition bel(φ) will be satisfied *iff* φ is entailed by the agent's belief base. Now, as our goals stated, we want to be able to reason about the mental models of other agents. Therefore, we must be able to indicate *which* agent's mental state should be queried, by annotating the mental literal with an agent selector.

4.2.1 Agent Selector

The agent selector represents a selection of agents that the programmed agent knows about. The agent itself can be part of that selection. Conceptually, it is a list of *agent expressions*:

$$\mathcal{AS} = \{\mathcal{AE}_1, \mathcal{AE}_2, \ldots, \mathcal{AE}_n\}$$

where $\mathcal{AE}$ is an agent expression. Such an agent expression can be one of the following:

- An agent's literal name. For example, `maker`.
- A variable. This is a variable that needs to be substituted by a substitution resulting from earlier mental literal queries in the mental state condition.
- A quantor. Quantors are described in Section 5.6.1.

4.2.2 Annotating the mental literal

Since the GOAL language was inspired by logic programming, it is desirable to have a syntax that closely matches a syntax that is conventional in logic programming, so a syntax like[1]

$$\text{bel}(AS, \varphi)$$

would be a nice choice, where *AS* is the agent selector, and φ the KR expression. But, in order for the grammar to be backwards compatible as our constraint 2 dictates, it should be possible to omit the *AS*. However, the parser then cannot differentiate between the case with *AS* and the case without *AS*, because the syntax of φ might overlap that of *AS*. The mental literal on line 1 in Figure 4.2 shows a conventional local belief query for the fact `somefact`. Line 2 shows a belief query for the fact `somefact` in the mental model of `agent1`'s mental state. But line 2 might just as well have been a local belief query for the conjunction of facts `agent1` and `somefact`. There are two possible approaches to solve this problem:

1. surround the *AS* with delimiters

[1] 'bel' is used throughout this section to illustrate the issue of syntax choice, but that issue holds for all four mental state operators

```
GOAL
bel(somefact)
bel(agent1, somefact)
```

Figure 4.2: Grammar conflict

2. make the presence of *AS* required, and in the case of a local query, use `self` to indicate as much.

Approach 1 does not really solve the problem of conflicting syntax with the syntax of φ, because even with delimiters there is a possibility of overlap with the syntax of φ.

Approach 2 does not solve the problem because it violates criterium 2; requiring something like `self` for the local queries breaks code that is written for single agent situations. Besides, it is counter-intuitive to have to write `self` every time when you are just programming a single agent.

What if we moved the *AS* outside the parenthesis of the operator?

```
GOAL
bel[agent1, B, agent2](p)
```

Figure 4.3: Agent selector between operator and parentheses

This is better in terms of separation from the propositional content. However, it still leaves a reader 'guessing' after its semantics. There is no strong intuitive link which suggests "this looks like the belief base of `agent1`, some variable agent `B` and `agent2`, are queried for `p`". Moving the agent selector to after the parentheses does read naturally, but it requires something to link the operator and

```
GOAL
bel(p) [by agent1, B, agent2]
```

Figure 4.4: Agent selector after parentheses, with 'by'

the agent selector. Prepositions from natural language like 'by' or 'of' complicate programming and the syntax, but leaving it out means losing the natural link between the operator and the agent selector.

We could put the agent selector in front of the operator. Some examples of this format are listed in Figure 4.5. This reads somewhat naturally as a sentence of the form `subject verb object`, the

```
GOAL
[agent1, B, agent2] bel(p)
agt:agent1, B, agent2 bel(p)
agent1, B, agent2 @ bel(p)
```

Figure 4.5: Agent selector before operator

subject being the conjunction of agent expressions, the verb being the operator and the object being the propositional content; “agent1, some agent B and agent2 believe p”. In the second version, the `agt:` makes it clear that agent expressions will follow, improving the readability. The syntax could benefit from more coherency, so enclosing the agent expressions in brackets to form the agent selector, and joining the agent selector with the operator with some sort of connector symbol improves readability. The versions of lines 1 and 3 can be generally specified as shown in Figure 4.6. In this syntax, the

GOAL

```
[agent1, B, agent2].bel(p)

% (brackets are optional if agent selector has only one agent expression):
X.bel(p)
agent3.a-goal(p)
allother.goal(p)
```

Figure 4.6: A proposed syntax for the agent selector

agent selector and connecting symbol can be easily omitted which would yield the semantics of a local query. This makes the grammar backwards compatible, satisfying criterium 2. The ordering of the language constructs resembles the natural language expression of it’s semantics, which helps to satisfy criterium 1. It allows expressing statements about beliefs and goals from mental models of other agents, by using literal names, variables and quantors to annotate the operators, which satisfies criterium 3.

4.2.3 Determining the annotation syntax preference

The syntax proposed in Figure 4.6 satisfies the criteria we stated earlier, but this does not mean we have automatically determined the most optimal choice of syntax. For some, the intuitiveness of the syntax may depend on other factors than the natural language association. For example, choosing a period ‘`.`’ as connector symbol establishes an intuitive link with Object Oriented Programming. Whether this is desirable is debatable, since it depends on the programmer’s preference and programming background. Similarly, some programmers might prefer having the agent selector in a different position.

Because such preferences do not follow from the above reasonings alone, and because the programmer is so important in this decision, a small survey was conducted to poll the preference of the potential users.

Survey setup

The objective of the survey was to determine the preference of potential users of the GOAL programming language with respect to the syntax of a mental model query. The survey had the form of a qualitative questionnaire, in which several potential users were asked to indicate their preferred syntax of the mental model query. Three options for the location of the agent selector were considered;

A. `bel(X, p)`

B. `X.bel(p)`

C. `bel[X](p)`

where `X` represents the agent selector. For option 4.2.3 three variants of the connector symbol were selected for consideration;

1. `X.bel(p)`
2. `X:bel(p)`
3. `X@bel(p)`

Results The results from the questionnaire show a general preference for option B, mostly because of the familiar OO-link. Option A was often seen as ambiguous and C as incoherent and unclear.

Of the variants on option B, variant 1 was deemed most preferable, also because of its strong association with the OO-style syntax of popular programming languages such as Java and C++.

From this survey it was decided to select the syntax as shown in Figure 4.6 for the mental model query.

4.2.4 Changing the grammar

Now that the syntaxes for the mental model queries and the `send` action have been determined, these needed to be implemented in the GOAL grammar.

The existing grammar had to be changed in several places. First, the `mentalLiteral` was extended to accept an optional `agentSelector`. This `agentSelector` has the grammar shown in Figure 4.7.

```
Grammar
agentSelector
  :
     agentExpression
  |
     '[' agentExpression (',' agentExpression)* ']'
  ;
```

Figure 4.7: Grammar of the `agentSelector`

`agentExpression` is one of the possible quantors (`SOME`, `SOMEOTHER`, `ALL`, `ALLOTHER`), `SELF`, a variable, or a constant.

To ensure consistent handling of mental literals whether an `agentSelector` was given or not, the parser automatically inserts an `agentSelector` with a single `SELF agentExpression` if no `agentSelector` is given for the mental literal. This follows the decided semantics that if no `agentSelector` is given, the query is performed on the agent's own mental state in stead of on a mental model.

The parser performs a semantic check to insure that inconsistent combinations of `agentExpressions` are not allowed. If a quantor is used, it can be the only `agentExpression` in the `agentSelector`.

Now, the `agentSelector` could be integrated into the grammars for the `mentalLiteral`, as shown in Figure 4.8.

```
Grammar
mentalLiteral
  :
    ( agentSelector '.')? mentalAtom
  | NEGATION LBRACKET ( agentSelector '.')? mentalAtom RBRACKET
  ;
```

Figure 4.8: Grammar of mental literal

The `send` action is a new type of internal action. It is added to the grammar alongside the other internal actions such as `adopt`, `drop`, `insert` and `delete`. The new (partial) grammar of `action` is given in Figure 4.9.

```
Grammar
action
  :
  (
    ... // other internal actions
  |
    'send' '(' destination = selector ',' mood = sentenceMood
    { if (mood == null) { mood = SentenceMood.INDICATIVE; } }
    { PrologTerm t = ParsePrologConjunction();
      act = new SendAction(checkMessageDestination(destination),PrologDBFormula(t)); }
    ')'
  )

sentenceMood : ':' | '!' | '?' ;
```

Figure 4.9: Grammar of the `send` action and the `sentenceMood`

Here it can be seen that if the sentenceMood is not given, the indicative mood is automatically assumed. This is conform the decision on backwards compatibility.

Also, a semantic check is made on the message destination. This `checkMessageDestination` insures that, in addition to the checks on `agentSelector`s mentioned above, the quantors `SOME` and `SOMEOTHER` are not used. This is because sending a message to 'some' agent is not allowed.

4.3 Mental models

A mental model is like a mental state, but with one difference: a mental state has a percept base and a mailbox, which a mental model does not. Otherwise a mental model is just a mental state. Or, a mental state is just a mental model, with an added percept base and mailbox. In this view it was decided to move all functionality concerning the belief base and the goal base from the `MentalState` class to the new `MentalModel` class. The `MentalState` class will contain all methods and fields related to the percept base and mailbox. It also holds a mapping of agent names (`String`s) to `MentalModel`s.

Upon construction of an agent, and thus its mental state, a `MentalModel` is initialized and added to the mapping of `MentalModel`s under the agent's own name. All operations on the `MentalState` that affect or refer to the belief- or goal bases are forwarded to the agent's own `MentalModel`.

4.3.1 Performing a mental model query

The main method for performing a query of a mental literal in the mental state is altered to take the `agentSelector` into account. First, the `agentSelector` is resolved. This means that the `agentSelector` is evaluated into a list of agent names. The list of currently known agent names and the name of the running agent is passed to the `resolve` method of the `agentSelector`, which uses these data to resolve the agent selector into a list of agent names. For each of the `agentExpressions` in the `agentSelector`, the `resolve` method evaluates which names are to be added to the resulting list. For each of the resulting agent names, the corresponding `MentalModel` is retrieved and the query of the mental literal is performed on that `MentalModel`. The result of one mental model query is substituted in the query of the next, in much the same way as is done in the conjunction of mental literals in a mental state condition.

4.4 Communication

Communication, on the implementation level, involves the sending and receiving of messages. When a `send` action is performed, a `Message` object is constructed, which is a simple container object containing the name of the sending agent, the name of the receiving agent, the mood of the message and the message content itself. One `send` action execution may send messages to multiple agents. For every agent that the `agentSelector` resolved to, a `Message` object is constructed and sent.

4.4.1 Sending

First, depending on the activated middleware, the `Message` is appended to the receiving agent's message in queue. Then the sending agent's mailbox is updated with the fact that this message was sent. This involves inserting a fact into a database of a particular knowledge representation language. This fact has the form `sent(<recipient>, <message>)`. Here, `<recipient>` is the name of the receiving agent. The `<message>` must also contain the mood operator, but this poses a problem. The mood operator symbols used in the GOAL language ('`:`', '`!`' and '`?`') are usually already part of the KR language. Simply placing the mood operator symbol in front of the message sentence, as is done in the GOAL language will likely result in a parse error when inserting the fact in the mailbox. For example, the '`!`' symbol is used in the Prolog language as the 'cut' operator. These syntax conflicts also exist when querying the mailbox in the GOAL code.

To avoid these syntax conflicts, instead of prepending these mood operator symbols to the message sentence, the sentence is placed in a predicate that represents the mood. For the imperative mood, this is `imp(..)` and for the interrogative mood this is `int(..)`. Sentences with an indicative mood do not get a predicate like the imperative and interrogative sentences do. So querying the mailbox for an indicative message is done simply by not placing the sentence in a predicate. See Figure 4.11 for an example.

```
GOAL
program {
  if bel(sent(grinder, ?canMake(grinder, _)))    % This will not parse into
                                                 % proper Prolog
    then adopt(handleResponse(grinder)).


  if bel(sent(grinder, int(canMake(grinder, _)))) % This does work
    then adopt(handleResponse(grinder)).
}
```

Figure 4.10: Mood operators in the mailbox

```
GOAL
program {
  if bel(agent(A), me(Me), not(sent(A, canMake(Me, [grounds]))))
    then send(A, :canMake(Me, [grounds])).
}
```

Figure 4.11: Querying the mailbox for an indicative message

4.4.2 Receiving

Messages are passed between agents asynchronously, in the sense that when an agent sends a message to another agent, the receiving agent does not immediately process it into its mailbox. Communication is therefore also non-blocking. Incoming `Message` objects are placed in a message in queue. At the beginning of an agent's run cycle, the messages in the message in queue are processed. This processing consists of three steps:

1. If the receiving agent does not have a mental model for the sending agent, it creates one and adds it to the mapping of agent names to mental models.
2. The mental model of the sending agent is retrieved from this mapping and updated. The updating operations depend on the mood of the message.
3. The receiving agent's mailbox is updated with a `received(..)` fact in the same way as was described for the `sent` fact in the above section.

From this point onwards in the agent's run cycle, the received messages are available through mailbox queries and mental model queries.

4.4.3 Optimization

For every message that is received, operations on the mental models are performed to insure that the mental models reflect the best up to date view of the other agent's mental state. There may be cases, however, where an agent's program contains no mental model queries. In such a case, the mental models will never be referenced. It would therefore be unnecessary to maintain them. To save the overhead of these updating operations they are only performed if the program contains at least one mental model query. This is checked at parse time by the parser.

4.5 Conclusion

The implementation of the communication constructs and mental models involved changing the grammar of the GOAL language as well as changing the GOAL interpreter code.

The changes to the grammar required language design decisions. Language design decisions are always difficult to make, because of their subjective nature. Therefore I conducted a survey to determine the preferred syntax of the mental model query. The resulting syntax preference was implemented in the new grammar.

Mental models were implemented by restructuring the mental state and the databases of an agent in the interpreter code. To enable querying of mental models, I implemented routines for resolving `agentSelector`s and changed the way queries on mental literals are handled.

The `send` action was implemented as an internal action. For this the syntax of the action was added to the GOAL grammar and the handling of `Message` objects was integrated into the agent's run cycle handling. Also, automatic updates of the agent's mailbox and mental models were implemented.

The fact that mental models are not referenced if an agent's program contains no mental model queries left room for optimization. This optimization was realized by automatically disabling the creation and updating procedures of mental models.

This implementation effort resulted in that GOAL agent programs can make use of communication using agent selectors and sentence moods, and use mental models to reason about the mental states of other agents. Chapter 5 will discuss examples of communication-enabled GOAL programs.

Chapter 5

Communication

5.1 Introduction

In a multi-agent system, it is useful for agents to communicate about their beliefs and goals. Agents may have only a partial view on the environment, and by communicating, agents may inform each other about parts they but other agents cannot perceive. Agents may also use communication to share goals and coordinate the achievement of these goals.

This chapter will explain how communication works in the GOAL platform, and how to program communicating GOAL agents. First, the organization of a multi-agent system by means of a MAS file is explained. Next, the communication primitive and the handling of messages is discussed. Sections 5.5 and 5.6 go into more detail on different types of messages and how to address other agents. Finally, an example multi-agent system is presented to demonstrate usage of communication to have agents coordinate their actions.

5.2 Multi-Agent Systems

GOAL facilitates the development and execution of multiple GOAL agents. These agents may or may not be associated with entities from the environment. Agents can be launched when the multi-agent system is launched, or when an entity in the environment is born.

Agents in a multi-agent system (MAS) can communicate with each other. Communication is essential in situations where agents have different roles and need to delegate actions to appropriate agents, or when agents with conflicting goals operate in the same environment space and need to coordinate their actions to prevent deadlocks and inefficiencies.

This section explains how to define a MAS, and how to use communication in agent programs.

5.2.1 Example MAS

Throughout this chapter we will be exploring the concepts of multi-agent systems and communication guided by an example MAS. This example MAS is described below.

Coffee domain

The Coffee domain is a multi agent system in which a coffee maker and a coffee grinder work together to brew a fresh cup of coffee. Optionally, a milk cow can provide milk for making a latte. To make coffee the coffee maker needs *water* and *coffee grounds*. It has water, and *coffee beans*, but not *ground* coffee. Grinding the beans is the task of the coffee grinder. The coffee grinder needs beans, and produces grounds. The programs of the coffee maker and the coffee grinder are listed in Figures A.1 and A.2, respectively.

The agents are designed in such a way that they know which ingredients are required for which products. They know what they can make themselves, but they don't initially know what the other agents can make. This is where communication comes in.

Figure 5.1 lists the agent program for the coffee maker agent. See Figure A.1 for a version with comments.

The knowledge section clearly reflects the agent's knowledge of which ingredients are necessary for which products. The beliefs section holds the agent's beliefs in what it can make. In this case, the

GOAL

```
main: coffeeMaker {
    knowledge {
        requiredFor(coffee, water).
        requiredFor(coffee, grounds).
        requiredFor(espresso, coffee).
        requiredFor(grounds, beans).

        canMakeIt(M, P) :- canMake(M, Prods), member(P, Prods).
    }
    beliefs {
        have(water). have(beans).
        canMake(maker, [coffee, espresso]).
    }
    goals {
        have(coffee).
    }
    program {
        if goal(have(P)) then make(P).
    }
    actionspec {
        make(Prod) {
            pre { forall(requiredFor(Prod, Req), have(Req)) }
            post { have(Prod) }
        }
    }
    perceptrules {
        if bel(agent(A), not(me(A)), not(canMake(A, _))) then sendonce(A, ?canMake(A, _)).
        if bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prod))
            then sendonce(A, :canMake(Me, Prod)).
        if bel(received(Sender, canMake(Sender, Products))) then insert(canMake(Sender, Products))
            + delete(received(Sender, canMake(Sender, Products))).

        if bel(agent(A), received(A, have(X))), not(bel(have(X))) then insert(have(X)).
        if goal(have(P)), bel(requiredFor(P, R), not(have(R))),
           bel(canMakeIt(Me, R), me(Me)) then adopt(have(R)).
        if goal(have(P)), bel(requiredFor(P, R), not(have(R))),
             bel(canMakeIt(Maker, R), not(me(Maker))) then sendonce(Maker, !have(R)).
        if bel(agent(Machine), received(Machine, imp(have(X))), have(X))
            then sendonce(Machine, :have(X)).
    }
}
```

Figure 5.1: The Coffee Maker

`maker` can make `coffee` and `espresso`. The goal section states this agent's mission: having `coffee`. Note that this describes a goal *state* (`coffee` being available), not an action (like 'making `coffee`'). Also note that the perceptrules section contains all communication related action rules, meaning that every round all instances of these action rules are executed. This is discussed in Section 5.7.

The coffee domain assumes the following, for simplicity's sake:

1. Resources (like `water`, `beans`, `grounds` and `coffee`) cannot be depleted.

2. The agents share the resources in the sense that if one agent has a resource, all agents do. But there is no environment, so agents cannot *perceive* changes in available resources; they have to communicate this. For example if the coffee grinder makes `grounds`, it will thereafter believe `have(grounds)`, but the coffee *maker* will not have this belief until it gets informed about it.

5.2.2 MAS Files

A multi-agent GOAL system needs to be specified by means of a *MAS file*. A MAS file in GOAL is a recipe for running a multi-agent system. It specifies which agents should be launched when the multi-agent system is launched and which GOAL source files should be used to initialize those agents. GOAL allows for the possibility that multiple agents instantiate a single GOAL agent file. Various features are available to facilitate this. In a MAS file one can associate multiple agent names with a single GOAL file. Each agent name additionally can be supplied with a list of optional arguments. These options include the number of instances of an agent, indicated by `#nr`, that should be launched. This option is available to facilitate the launching of large numbers of agents without also having to specify large numbers of different agent names.

A MAS file is a recipe for executing a multi-agent system. The GOAL interpreter uses these files to launch a multi-agent system and an environment. A MAS file should provide the information to locate the relevant files that are needed to run a multi-agent system and the associated environment. A MAS file has the following format:

masprogram	::=	[*envdesc*] *agentfiles launchpolicy*
envdesc	::=	**environment:** *path* .
path	::=	any valid path to a file in quotation marks
agentfiles	::=	**agentfiles:** { *agentfile* {, *agentfile*}* }
agentfile	::=	*path* [*agentparams*] .
agentparams	::=	[*nameparam*] $\mid$ [*langparam*] $\mid$
		[*nameparam* , *langparam*] $\mid$
		[*langparam* , *nameparam*]
nameparam	::=	**name =** *id*
langparam	::=	**language =** *id*
launchpolicy	::=	**launchpolicy:** { { *launch* $\mid$ *launchrule* }* }
launch	::=	**launch** *agentbasename* [*agentnumber*] : *agentref* .
agentbasename	::=	$*$ $\mid$ *id*
agentnumber	::=	[*number*]
launchrule	::=	**when** *entitydesc* **do** *launch*
entitydesc	::=	[*nameparam*] $\mid$ [*typeparam*] $\mid$ [*maxparam*] $\mid$
		[*nameparam* , *typeparam*] $\mid$ [*typeparam* , *nameparam*] $\mid$
		[*maxparam* , *typeparam*] $\mid$ [*typeparam* , *maxparam*]
typeparam	::=	**type =** *id*
maxparam	::=	**max =** *number*
id	::=	and identifier starting with a lower-case letter
num	::=	a natural number

A MAS program consists of three sections:

1. an *environment description*-section that defines the connection to one environment interface,
2. a section of *agent files* that defines a list of GOAL-files, and
3. a *launch policy*-section that defines a policy how and when to instantiate agents from the GOAL-files.

An environment description defines a connection to one environment interface. That environment interface is supposed to be a jar file that conforms with EIS[1].

Example:

```
environment: "elevator.jar" .
```

The agent files define the set of GOAL-files that are to be used. Those GOAL-files are then referenced by the agents. You can simply define a GOAL-file like this:

```
agentfiles: {
  "elevatoragent.goal" .
}
```

[1]Environment Interface Standard

The reference label for the agents would then be `elevatoragent`, which is the file name without its file extension. However you can define the reference label yourself by using *agent parameters*. You can also define the knowledge representation language this way. Here is an example:

```
agentfiles: {
  "elevatoragent1.goal" [language=swiprolog,name=file1] .
  "elevatoragent2.goal" [language=pddl,name=file2] .
}
```

This defines two agent files. The first is referenced by the label `file1` and uses SWIProlog as KR language. The second is referenced by `file1` and uses PDDL[2].

The final section contains the *launch policy*. The launch policy consists of a list of *launches* and *launch rules*. A launch is applied before running the MAS and instantiates agents that do not have a connection to the environment. An example:

```
launchpolicy {
    launch elevator:file1 .
}
```

This launches a single agent and uses the agent file that is labelled by `file1`. It also uses the identifier `elevator` as the base name for the generation of unique agent names. You can also instantiate several agents with one launch:

```
launchpolicy {
    launch elevator[3]:file1 .
}
```

This launch would instantiate three agents. The agent names would be `elevator`, `elevator1`, and `elevator2`.

A launch rule on the other hand is applied to instantiate an agent or agents when the environment contains an entity that is not associated with an agent, called a *free entity*. This happens when the environment initializes a new elevator carriage, for example. A launch rule is triggered by the creation of an entity in the environment. Special conditions can be added on the type of event or trigger. This is a very simple launch rule:

```
launchpolicy {
    when entity@env do launch elevator:file1 .
}
```

Its interpretation is: when there is a free entity create an agent with the base name `elevator` from `file1` and associate it with the entity.

You can also do something useful with the base name:

```
launchpolicy {
    when entity@env do launch *:file1 .
}
```

[2]PDDL support is under development and is not yet available

The asterisk means that the name of the entity as provided by the environment is used as the base name for the agent.

Of course you can also instantiate several agents:

```
launchpolicy {
    when entity@env do launch elevator[3]:file1 .
}
```

This would instantiate three agents and associate them with one and the same entity. So if the entity would perceive something, all three agents would receive that percept. If any of those agents performs an environment action, it will be performed by that entity.

Launch rules can be conditional on the type, amount and name of the entity/entities:

```
launchpolicy {
    when [type=type1]@env do launch elevator:file1 .
  }
```

This would only launch an agent, when the type of the new entity is `type1`.

You can also restrict the amount of instantiated agents:

```
launchpolicy {
    when [type=type1,max=20]@env do launch elevator:file1 .
  }
```

This launch rule would only be applied at most 20 times.

There is also a name parameter:

```
launchpolicy {
    when [name=elevator1]@env do launch elevator:file1 .
  }
```

This would only be applied if the new entity has the name `elevator1`.

5.2.3 Automatic `agent` and `me` fact generation

In many practical multi-agent situations, the agents that are to be launched in the MAS and their names are not known during programming, but are determined in the MAS file, as described in Section 5.2.2. Also, in some MASs, agents may come and go dynamically during the lifetime of a MAS. It is therefore not always possible or practical to hard code the known agents in the belief base.

Instead, GOAL automatically inserts these `agent` facts in the belief base whenever a new agent enters the MAS, and upon launch of an agent, it populates the belief base with an `agent` fact for each existing agent (including itself). An agent program(mer) can thus assume that, at any time, `bel(agent(X))` will result in a substitution for `X` of each existing agent.

To give an agent knowledge of its own name and thus the ability to distinguish itself from the other agents amongst the `agent` facts, a special `me` fact is inserted into its belief base. It has the form `me(<agentname>)` where `<agentname>` is the name of the agent, as determined by the launch policy.

It is therefore not necessary to specify or maintain a list of existing agents, or to hard code the agent's name in the program.

Unless an agent wants to actively ignore some agent, it is unwise to `delete agent` facts from the belief base, and should therefore be avoided.

5.2.4 Example MAS file

A minimal MAS file without environment would look like Figure 5.2. This would start a MAS without

GOAL MAS file

```
agentfiles {
  "agent.goal".
}

launchpolicy {
  launch agent1:agent.
}
```

Figure 5.2: A minimal MAS file

an environment, with one agent named `agent1` whose agent program is loaded from file `agent.goal`. This agent's belief base will contain the following facts:

```
beliefs {
  ... % other facts
  agent(agent1).
  me(agent1).
}
```

A MAS file for the coffee domain would be as shown in Figure 5.3

GOAL MAS file

```
agentfiles {
    "coffeemaker.goal".
    "coffeegrinder.goal".
}

launchpolicy {
    launch maker:coffeemaker.
    launch grinder:coffeegrinder.
}
```

Figure 5.3: A MAS file for the coffee domain MAS

After launch of the agents, the coffee maker's belief base would look like this:

```
beliefs {
  have(water). have(beans).
  canMake(maker, [coffee, espresso]).
  agent(maker).
  agent(grinder).
  me(maker).
}
```

A more complex situation is given in Figure 5.4.

GOAL MAS file

```
environment: "environments/elevatorenv.jar".

agentfiles {
    "goalagents/elevatoragent.goal" [name=elevatorfile] .
    "goalagents/managingagent.goal" [name=managerfile] .
}

launchpolicy {
    launch manager:managerfile .
    when [type=car,max=1]@env do launch elevator1:elevatorfile .
    when [type=car,max=1]@env do launch elevator2:elevatorfile .
    when [type=car,max=1]@env do launch elevator3:elevatorfile .
}
```

Figure 5.4: A more complex MAS file

This example uses relative paths to the files and labels to reference those files. One elevator agent will be launched and associated with each entity in the environment of type `car` (at most three times).

After all three elevator agents have been launched, the belief base of `elevator2` will look like

```
beliefs {
  ... % other facts
  agent(manager).
  agent(elevator1).
  agent(elevator2).
  agent(elevator3).
  me(elevator2).
}
```

5.3 Communication

Communication in the current implementation of GOAL is based on a simple “mailbox semantics”. Messages received are stored in an agent’s mailbox and may be inspected by the agent by means of queries on special, reserved predicates `sent(`*agent*,*msg*`)` and `received(`*agent*,*msg*`)` where *agent* denotes the agent the message has been sent to or received from, respectively, and *msg* denotes the content of the message expressed in a knowledge representation language.

5.4 Send Action and Mailbox

The action `send(AgentName, Poslitconj)` is a built-in action to send `Poslitconj` to the agent with given `AgentName`. `Poslitconj` is a conjunction of positive literals. `AgentName` is an atom with the name of the agent as specified in the MAS file. Messages that have been sent are placed in the mailbox of the sending agent, as a predicate of the form `sent(AgentName, Poslitconj)` (note the '`t`' at the end of `sent`). The message is sent over the selected middleware to the target agent, and after arrival the message is placed there in the form `received(SenderAgentName, Poslitconj)` where `SenderAgentName` is the name of the agent that sent the message. Depending on the middleware and distance between the agents, there may be delays in the arrival of the message. In the current implementation of GOAL messages are supposed to always arrive.

5.4.1 The `send` action

To illustrate the working of the `send` action, let's consider a simple example multi-agent system consisting of two agents, *fridge* and *groceryplanner*. Agent *fridge* is aware of it's contents and will notify the *groceryplanner* whenever some product is about to run out. The *groceryplanner* will periodically compile a shopping list. At some point, the fridge may have run out of milk, and takes appropriate action:

```
program {
    ...
    if bel(amountLeft(milk, 0)) then send(groceryplanner, amountLeft(milk, 0)).
    ...
}
```

At the beginning of its action cycle, the *groceryplanner* agent gets the following fact inserted in its message base.

```
    received(fridge, amountLeft(milk, 0)).
```

The received messages can be inspected by means of the `bel` operator. In other words, if an agent has received a message *M* from sender *S*, then `bel(received(`*S*, *M*`))` will be true; the agent believes it has received the message. This also holds for `bel(sent(`*R*, *M*`))`, where *R* is the recipient of the message. This way, the *groceryplanner* can act on the received message:

```
program {
    ...
    if bel(received(fridge, amountLeft(milk, 0))) then adopt(buy(milk)).
}
```

5.4.2 Mailbox management

In contrast with the percept base, mailboxes are not emptied automatically. This means that once a message is sent or received, that fact will remain in the message base, even after execution of the above program rule. The consequence of this is that the next action cycle, the *fridge* may again select the shown program rule, sending the same message again, over and over. Also, the *groceryplanner* will keep selecting this program rule.

We have to take action to prevent this. There may be some special cases in which it is preferred to leave the message in in the mailbox, for example if the message contains some message counter, so you can review the whole message history. Otherwise it is possible that a new message containing the same content sent to the same recipient will not be seen as a new message. So, we need to remove the `received` when we process them. For this an internal action is added to the action rule.

```
    if bel(received(fridge, amountLeft(milk, 0)))
        then adopt(buy(milk)) + delete(received(fridge, amountLeft(milk, 0))).
```

If the *fridge* sends this message only once, this program rule will be selected only once.

The coffee maker agent from Section 5.2 also gives an example of this:

```
    % process information from other agents on what they can make
    if bel(received(Sender, canMake(Sender, Products)))
        then insert(canMake(Sender, Products)) + delete(received(Sender, canMake(Sender, Products)))
```

The logic is slightly different for the sender, because if it would remove the `sent` fact it would lose the belief that it has already notified the *groceryplanner*, and send the message again. Instead it can use this information to prevent repeatedly sending the same message:

```
    if bel(amountLeft(milk, 0), not(sent(groceryplanner, amountLeft(milk, 0))))
        then send(groceryplanner, amountLeft(milk, 0)).
```

The sendonce action

Because the above leads to verbose programming, GOAL offers a variant of the `send` action; the `sendonce` action. The syntax is the same as that of `send`, but the semantics are such that the message is sent only if there is no `sent` fact for that message (and receiver(s)) in the mailbox. Writing

```
  if bel(agent(A), fact(P)) then sendonce(A, fact(P)).

  % if some machine seems to need a product, tell it we have it
  if bel(agent(Machine), received(Machine, imp(have(P))), have(P))
      then sendonce(Machine, have(P)).
```

is short for

```
  if bel(agent(A), fact(P), not(sent(A, fact(P)))) then send(A, fact(P)).

  % if some machine seems to need a product, tell it we have it
  if bel(agent(Machine), received(Machine, imp(have(P))),
      have(P), not(sent(Machine, have(P)))) then send(Machine, have(P)).
```

This means that if the `sent` fact is deleted from the mailbox, the message may henceforth be sent again by the `sendonce` action.

5.4.3 Variables

In GOAL programs, the use of variables is essential to writing effective agents. Variables can be used in messages as expected. For example, a more generic version of the *fridge*'s program rule would be

```
    if bel(amountLeft(P, N), N < 2, not(sent(groceryplanner, amountLeft(P, N))))
        then send(groceryplanner, amountLeft(P, N)).
```

Note that this will eventually send one message for every value of N where $N < 2$.

Recipients and senders can also be variables in the mental state condition. Example:

```
    % This isn't an argument; it's just contradiction!
    % - No it isn't.
    if bel(received(X, fact)) then send(X, not(fact)).

    % http://en.wikipedia.org/wiki/Marco_Polo_(game)
    if bel(received(X, marco)) then send(X, polo).
```

This is especially useful in situations where you don't know who will send the agent messages, as with the coffee domain example:

```
    % answer any question about what this agent can make
    if bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prods))
      then sendonce(A, canMake(Me, Prods)).
```

For any agent A it has received a question from, it will answer its question.

Closed actions

In order for any action in GOAL to be selected for execution, that action must be closed, meaning that all variables in the action must be bound after evaluation of the mental state condition. As a consequence, *messages* must be closed as well, in order to make the action executable.

5.5 Moods

GOAL agents are goal-oriented agents who have their goals specified declaratively. Up until now all examples have shown communication of an agent's *beliefs*. Every message was a statement about the sender's beliefs regarding the content. Given GOAL's goal-orientedness, it would be useful to be able to not only communicate in terms of beliefs but also in terms of *goals*. This way GOAL agents can tell other agents that they have a certain goal.

In natural language communication, such a *speech act* is often performed by *uttering* a *sentence* in a certain *mood*. This mood can be *indicative* ('The time is 2 o'clock'), *expressive* ('Hurray!!'), *declarative* ('I hereby declare the meeting adjourned').

In GOAL, the execution of the send action is the *uttering*, the message content is the *sentence*. The *mood* is indicated by prefixing the message content with a *mood operator*. GOAL distinguishes three moods listed in Figure 5.5.

Mood	operator	example	NL meaning
INDICATIVE	:	`:amountLeft(milk, 0)`	“I’ve run out of milk.”
DECLARATIVE	!	`!status(door, closed)`	“I want the door to be closed!”
INTERROGATIVE	?	`?amountLeft(milk, _)`	“How much milk is left?”

Figure 5.5: GOAL message moods

In the case of the indicative mood the mood operator is optional. In other words, in absence of a mood operator, the indicative mood is assumed. That means that all examples in Section 5.4 were implicitly in the indicative mood.

Using these mood operators, GOAL agents can be more GOALish in their communication. For example, if the coffee maker or coffee grinder needs a resource to make something but hasn’t have it, it can inform an agent that it believes *does* have it that it needs it:

```
% if we need a product but don’t have it, notify an agent that does have that we need it.
if goal(have(P)), bel(requiredFor(P, R), not(have(R)), canMakeIt(Maker, R)) then send(Maker, !have(P)).
```

Now for the receiving side of the communication. Moods of messages in the mailbox are represented as predicates, allowing for logic programming. An imperative is represented by the `imp` predicate, an interrogative mood by the `int` predicate. There is no predicate for the indicative mood in the mailbox. Using these mood predicates, we can inspect the mailbox for messages of a specific type. For example, to handle a message like the one above from the coffee maker, the coffee grinder can use this action rule:

```
% if some agent needs something we can make, adopt the goal to make it
if bel(received(_, imp(have(P))), me(Me), canMakeIt(Me, P)) then adopt(have(P)).
```

The coffee grinder will grind beans for whichever agent needs them, and another rule will make sure the correct agent is notified of the availability of the resulting grounds, so here a *don’t care* is used in place of the sender parameter.

The previous section mentioned that messages must be closed. There is one exception, which concerns interrogative type messages. These messages are like open questions, like, for example, “What time is it?” or “What is Ben’s age?”. These cannot be represented by a closed sentence. Instead, a *don’t care* can be used to indicate the unknown component. For example:

```
if not(bel(timeNow(_))) then send(clock, ?timeNow(_)).

if not(bel(age(ben, _))) then send(ben, ?age(ben, _)).

% ask each agent what they can make
if bel(agent(A), not(me(A)), not(canMake(A, _))) then sendonce(A, ?canMake(A, _)).
```

5.6 Agent Selectors

In many MASs agents may find themselves communicating with agents whose name they do not know beforehand. For example, the MAS might have launched 100 agents, who communicate with each other, using the `agent[100]:file1` syntax. Or if a message needs to be multicast or broadcast to multiple receivers. For these cases a more flexible way of addressing messages is needed.

5.6.1 `send` action syntax

The `send` action allows more dynamic addressing schemes than just the agent name, by means of an *agent selector*. The syntax of the `send` action and this agent selector is shown in Figure 5.6.

The first parameter to the `send` action (agent name in the previous sections) is called an *agent selector*. An agent selector specifies which agents are selected for sending a message to. It consists of one or more *agent expression*s, surrounded by square brackets. The square brackets can be omitted if there is only one agent expression.

Some examples of agent selectors:

```
% agent name
send(agent2, theFact).

% variable (Prolog)
send(Agt, theFact).

% message to the agent itself
send(self, theFact).

% multiple recipients
send([agent1, agent2, self, Agt], theFact).

% using quantor
% if we don’t know anyone who can make our required resource, broadcast our need
if goal(have(P)), bel(requiredFor(P, R), not(have(R)), not(canMakeIt(_, R)))
    then send(allother, !have(P)).
```

sendaction	::=	**send (** *agentselector* **,** [*moodoperator*] *Poslitconj***)**
moodoperator	::=	**:** | **!** | **?**
agentselector	::=	*agentexpression* |
		quantor | **[** *quantor* **]** |
		[*agentexpression* [**,** *agentexpression*]* **]**
agentexpression	::=	*label* | *variable* | **self**
quantor	::=	**all** | **allother**

Figure 5.6: Syntax of the `send` action

Agent Name

The agent name is the simplest type of agent expression, which we have already seen in Sections 5.4 and 5.5. It consists of the name of the receiving agent. If the KR language of the agent is Prolog, the agent name must start with a lowercase letter.

Example:

```
  send(alice, :hello).

  % using the square brackets to address multiple agents for one message
  send([alice, bob, charlie], :hello).
```

If the agent name refers to an agent that does not exist in the MAS, or has died, or is otherwise unaddressable, the message will silently be sent anyway. There is no feedback confirming or disconfirming that an agent has received the message. Only a reply, the perception of expected behaviour of the receiving agent, or the absence of an expected reply can confirm or disconfirm the reception of the message.

Variables

A variable type agent expression allows a dynamic way of specifying the message recipient. Sometimes the recipient depends on the agent's beliefs or goals or on previous conversations. The variable agent expression consists of a variable in the agent's KR language. If the KR language is Prolog, this means it must start with an uppercase letter. This variable will be resolved when the program rule's mental state condition is evaluated. This means that the mental state condition **must** bind all variables that are used in the agent selector. If an agent sector contains unbound variables at the time of action selection, the action will be deemed inapplicable.

Example:

GOAL

```
1  beliefs {
2    agent(john).
3    agent(mary).
4  }
5  goals {
6    informed(john, fact(f)).
7  }
8  program {
9    if bel(agent(X)), goal(hold(gold)), not(bel(sent(_, !hold(_))))  then send(X, !hold(gold)).
10
11   if goal(informed(Agent, fact(F))) then send(Agent, :fact(F)).
12
13   % This will never be selected:
14   if bel(something) then send(Agent, :something).
15 }
```

In this example, the program rule on line 9 contains the variable `X`, which has two possible substitutions: `[X/john, X/mary]`. This results in there being two *options* for the action: `send(john, !hold(gold))` and `send(mary, !hold(gold))`. The agent's action selection engine will select only one option for execution. This means that variables resolve to *one* agent name, and are therefore not suited for multicasting messages.

Quantors

Quantors are a special type of agent expression. They consist of a reserved keyword. There are three possible quantors: `all`, `allother` and `self`. When the `send` action is performed, the quantor is expanded to a set of agent names, in the following way:

- **`all`** will expand to all names of agents currently present in the belief base of the agent (including the name of the sending agent itself).
- **`allother`** will expand to all names of agents currently present in the MAS, with the exception of the sending agent's name.
- **`self`** will resolve to the sending agent's name. So using `self`, an agent can send a message to itself.

Sending a message addressed using a quantor will not result in the quantor being put literally in the mailbox. Rather, the actual agent names that the quantor resolves to are substituted, and a `sent(..)` fact is inserted for every agent addressed by the quantor. This has consequences for querying the mailbox using quantors. It is possible to test if a message has been sent to `all` agents, for example, by doing

```
  if bel(not(sent(all, fact))) then send(all, fact).
```

This will execute if the message has not been sent to all agents the sending agent believes to exist, so all substitutions of `X` in `bel(agent(X))`. This means that after sending of the original message, if new agents would join the MAS, this substitution would change (i.e. `agent(X)` facts would be added). Thus the above mental state condition would again be satisfied, because the message had not been sent to all agents. The semantics of the `all` and `allother` quantors in belief queries reflect the situation *at the time of querying*.

This is illustrated in the following code fragment, in which the `mailbox..` section reflects the mailbox contents at this time.

```
beliefs {
  agent(maker).
  agent(grinder).
  agent(auxilliarygrinder).
  me(maker).

  % the new agent that just joined the MAS
  agent(newagent).
}
mailbox {
  sent(grinder, imp(have(grounds))).
  sent(auxilliarygrinder, imp(have(grounds))).
}
program {
  % will execute again:
  if bel(not(sent(allother, imp(have(grounds))))) then send(allother, !have(grounds)).
}
```

5.6.2 The `agent` and `me` facts

In the previous section we have seen the use of variables in agent selectors, and how such a variable must be bound in the agent selector. In the example in that section the belief base was populated with 2 `agent(..)` facts, holding the names of the agents that agent believes to exist. Using this 'list' of agents, program rules can be constructed that send a message to agents that satisfy some criterium. For example, a way to send a request only to agents that are not busy could be;

```
if bel(agent(X), not(busy(X))) then send(X, !swept(floor)).
```

The `agent(X)` is crucial here, to get a substitution set for `X`, because `not(busy(X))` does not yield a substitution set for `X` by itself.

The `agents` and `me`

So the `agent` fact allows us to select a subset of all existing agents dynamically. An advantage of this is that it makes it possible to write 'dynamic' agent programs, meaning we can write *one* GOAL program for a MAS with multiple identical agents.

Let's reiterate the last example snippet:

```
if bel(agent(X), not(busy(X))) then send(X, !swept(floor)).
```

This will select one agent that is not busy, and send `!swept(floor)` to it. Recall that an `agent` fact is inserted for every existing agent, *including the agent itself*. Consequently, the agent whose program rule is given here, may send `!swept(floor)` to itself, as it is one of the `agent(X)`s. This may not be the intended behaviour. Suppose the behaviour should be that it only sends this imperative to *other* agents. We cannot use `allother` as agent selector, because, while it excludes the agent itself from the recipient list, it indiscriminately sends the message to *all* other agents, ignoring the selection we made in the mental state condition.

We need another way to distinguish between *other* agents and `this` agent. For this purpose, a special `me(..)` fact is inserted in an agent's belief base upon launch. It specifies the name of the agent. So, taking the example MAS from Figure 5.4, after launch of `elevator2`, its belief base consist of the following facts:

```
agent(manager).
agent(elevator1).
agent(elevator2).
agent(elevator3).
me(elevator2).
```

Now the elevator program can include a rule that sends a message to any other elevator agent, like so:

```
if bel(agent(Agt), me(Me), not(Agt=Me), not(Agt=manager)) then send(Agt, !service(somefloor)).
```

The whole point of this is that this program rule works for every elevator agent and so it is not necessary to make a GOAL program file for each agent in which the agents would be named explicitly[3]. Also, if the naming scheme or the number of the elevator agents were to be changed, the agent program would not have to be altered; only the MAS file would.

In the case of the coffee domain agents, it means that the coffee maker and the coffee grinder, which are both machines that can make something out of something, can have very similar programs, sharing action rules for production and capability exploration.

[3]with exception of the `manager`, but here we assume this to be a special agent that always has this name. If there were more `managers`, the belief clause would contain `bel(manager(Mgr), not(Agt=Mgr))`

5.7 `send` action processing

Action rules containing a `send(once)` action can be placed in the program rules section, which we have done so far, but also in the percept rules section. The way `send` actions are selected and executed differs between these sections. These differences and criteria are discussed below.

In the program rules The first strategy is placing the action rule in the program rules section, as we have done so far. Let's take a look at an example:

```
if goal(have(X)), bel(agent(A)) then sendonce(A, !have(X)).
```

Suppose there are three agents, and the agent has one goal `have(milk)`. The action selection mechanism will pool three options of this action to choose from for execution for this round, one `send` action for each agent. Only one will be selected and executed. Next round, only two options are pooled, etc.

It will take at least three rounds to notify all agents of the goal. To send this message to all agents at once we can use the `all` or `allother` agent selectors. But when we want to filter the agents to which the message will be sent we cannot do this.

In the percept rules Percept rules are similar to program rules except for two differences;

1. they cannot contain environment actions
2. all options of all percept rules are all executed every round

The second item is consequential for message sending. If the example action rule from the above paragraph was placed in the percept rule section, all three options would be executed in one round, so all three agents would be notified at the same time.

In many cases, it makes more sense to handle communication in a way that all possible messages are sent at once in stead of one per round. Often, the communication is a task that needs to be done, but should not interfere with the selection of an environment action. Examples of such communication tasks are answering incoming interrogatives, notifying agents of our goals and beliefs proactively, relaying messages, but also tasks that do not involve sending like handling incoming indicatives (inserting the content in the belief base).

5.8 Example: The Coffee Domain

In Section 5.2.1 the coffee domain was introduced. In this section the workings of the coffee maker and coffee grinder are analyzed in more detail.

As mentioned before, the agents coordinate their actions by communicating in several ways which are discussed below.

Capability exploration

The agents know what they can make themselves. This is represented as beliefs in the agent program. For the coffee maker, this look like:

```
beliefs {
  ...
  canMake(maker, [coffee, espresso]).
}
```

To find out what the other agents can make, the following action rules are used in the program:

```
  % ask each agent what they can make
  if bel(agent(A), not(me(A)), not(canMake(A, _)), not(sent(A, int(canMake(A, _)))))
    then send(A, ?canMake(A, _)).

  % answer any question about what this agent can make
  if bel(me(Me), received(A, int(canMake(Me, _)), canMake(Me, Prods))
    then send(A, :canMake(Me, Prods)) + delete(received(A, int(canMake(Me, _)))).

  % process answers from other agents
  if bel(received(Sender, canMake(Sender, Products)))
    then insert(canMake(Sender, Products)) + delete(received(Sender, canMake(Sender, Products))).
```

The first rule checks if there is an agent A, other than this agent, for whom this agent does not have any belief of what it can make, and to whom this agent has not already sent an *interrogative* to query it. If this is the case, send an *interrogative* message to ask which products that agent A can make. Note that not(me(A)) prevents A being bound to this agent, which would otherwise result in this agent asking itself what it can make. In this situation that would not happen, because not(canMake(A,_)) has the same effect, since this agent has a belief of what it can make (e.g. bel(me(Me), canMake(Me, _)) is true). Also recall that after execution of a send action, a sent fact is inserted in the mailbox.

The second rule handles such incoming interrogatives. It looks in the mailbox for received interrogative messages asking what this agent can make. It replies to the sender with an *indicative* message, indicating what it can make. Also, it removes the received message from the mailbox. This prevents this rule from being triggered repeatedly.

Finally these indicatives are handled in the third rule. The mailbox is queried for received *indicative* messages, containing the information about who makes what. If such a message exists, insert the information as a fact in the belief base. Also, the received message is removed from the mailbox to prevent repeated execution of this program rule for this message.

Production delegation

The coffee maker needs ground beans (grounds) to make coffee, but it cannot grind beans. But once it has found out that the coffee grinder *can* grind beans into coffee grounds, using the above program rules, it can request the grinder to make grounds by sending it an *imperative* message. This is represented more generically in the following action rule:

```
  % When we cannot make some product, try to find a maker for it
  if goal(have(P)), bel(requiredFor(P, R), not(have(R))), bel(canMakeIt(Maker, R), not(me(Maker)))
    then send(Maker, !have(R)).
```

When this agent has a goal to make some product P for which it needs a requirement R which it doesn't have, and it knows of a maker of R, it sends an imperative message to that maker. The message content is `!have(R)` (the `R` will be bound to some product at this point), indicating that this agent has a goal to `have R`.

When such an imperative message is received by an agent and it can make the requested product, it can adopt a goal to make it so:

```
if bel(received(A, imp(have(P))), me(Me), canMakeIt(Me, P))
  then adopt(have(P)).
```

Note that we did not remove the message from the mailbox. This is because this agent needs a record of who requested what. If we would remove the message, the information that an agent requested a product would have to be persisted by some other means.

Status updates

Once a product has been made for some other agent that requires it, that agent should be informed that the required product is ready. Agents in the Coffee Domain do not 'give' each other products or perceive that products are available, so they rely on communication to inform each other about that.

```
if bel(received(A, imp(have(P))), have(P))
  then send(A, :have(P)) + delete(received(A, imp(have(P)))).
```

Now we *do* remove the received message, because we have completely handled the case.

On the receiving side of this message, reception of such an indicative message `:have(P)` does not automatically result in the belief by this agent that `have(P)` is true. This insertion of the belief must be done explicitly[4].

```
% update beliefs with those of others (believe what they believe)
if bel(received(A, have(P)))
  then insert(have(P)) + delete(received(A, have(P))).
```

Pro-active inform

At any time, it may be the case that an agent sees an opportunity to inform an other agent about some fact if it thinks this agent would want to know that, without being asked. This may happen if it believes the other agent has some goal but it believes that this goal has already been achieved. It can then help the other agent by sending an indicative message that the goal state is achieved.

In the coffee domain example, if one machine believes that another machine needs some product, and it *has* that product available, then it will inform that agent of that fact:

```
% if some machine seems to need a product, tell it we have it
if bel(received(Machine, imp(have(X))), bel(have(X), not(sent(Machine, have(X))))
  then send(Machine, :have(X)).
```

[4]This is where we make a leap of faith. The other agent indicated *its* belief in `have(P)`. The only reason we copy this belief is because we trust that other agent.

The milk cow

The coffee domain example has a coffee maker and a coffee grinder. Suppose we now also want to make lattes. A latte is coffee with milk. To provide the milk, a cow joins the scene. The cow is empathic enough that it makes milk whenever it believes that someone needs it. The source code for the `milkcow` agent is listed in Figure A.3.

The generic way in which the `maker` and `grinder` agents were written has the effect that they need very little adjustment to start interacting with the `milkcow`. First, the `maker`'s beliefs are changed to reflect its new capability to make `latte`, and the recipe for `latte` is added to its knowledge:

```
beliefs {
  ...
  canMake(maker, [coffee, espresso, latte]).
}
knowledge {
  ...
  requiredFor(latte, milk).
  requiredFor(latte, coffee).
}
```

Then, the capability exploration routines will find out that the `milkcow` agent can make the required `milk`. Note that the `agent(milkcow)` fact will be added to the belief base automatically. The `grinder` needs no `milk`, and the `milkcow` needs no `grounds`, so adjustment of the `grinder` is not necessary.

Finally, to add the `milkcow` to the MAS, the MAS file is changed to include the new agent:

```
agentfiles {
    "coffeemaker.goal".
    "coffeegrinder.goal".
    "milkcow.goal".
}

launchpolicy {
    launch maker:coffeemaker.
    launch grinder:coffeegrinder.
    launch milkcow:milkcow.
}
```

5.9 Conclusion

In this chapter I discussed how communication in GOAL works from the programmer's perspective. First, an example multi-agent system was introduced.

Multi-agent systems are specified by MAS files, which define which agents are to be launched and how many, and what the names of the agents are. Each agent automatically gets an `agent` fact inserted in its belief base for each agent in the MAS, and a `me` fact to indicate the agent's own name. This way agents can refer to other agents and themselves in their GOAL programs.

Communication is done by executing a `send` internal action and follows a 'mailbox semantics', very similar to the communication semantics of 2APL [5]; the sent message is placed in the sending agent's mailbox as a `sent` fact, and in the receiving agent's mailbox as a `received` fact. Messages are addressed by means of agent selectors, which offer static and dynamic ways of selecting a subset of all agents in the MAS.

Because the mailbox holds no temporal information, some care needs to be taken when querying the messages in the mailbox to prevent unintended repeated execution of action rules. Best practices in dealing with these situations are presented. To address a common pattern in these dealings, the `sendonce` action is introduced, which does what `send` does but only if the message in question was not already sent to the same agents.

Finally the usage of the communication constructs presented in this chapter is demonstrated by implementing two agents of the example coffee domain. Using the presented communication constructs, a multi-agent system of communicating agents can be programmed.

Chapter 6

Mental Models

6.1 Introduction

In Chapter 3 it was argued that since GOAL is a declarative goal-oriented programming language, communication should also be programmable in a declarative goal-oriented manner. This means that agents should not only be able to communicate in terms of their beliefs but also in terms of their goals.

Chapter 5 describes how GOAL agents can communicate their beliefs and goals. But the mailbox semantics for handling incoming communication leaves room for improvement in terms of ease of programming. The mailbox semantics requires the programmer to actively process incoming messages and update the agent's mental state accordingly. While this offers flexibility, it requires programming effort on a low level of abstraction.

On a higher level of abstraction, more information can be extracted from the messages. Since the incoming messages convey information about the mental state of the sender, a receiving agent could directly derive facts about that mental state based on the *mood* and the *content* of the message. Instead of directly adopting the content of the message into the own mental state, the receiving agent can use these derived facts to automatically construct a model of the mental state of the sending agent.

In this chapter *mental models* are explored in detail. Mental models are models of the mental state of another agent. These models are constructed and maintained by observing incoming messages from other agents. The mental models can then be used by the agent (programmer) to reason about the beliefs and goals of another agent, in a similar fashion as an agent queries its own mental state.

First the issues with programming GOAL agents using mailbox semantics are investigated in the following section. Next the semantics of the mental model query is laid out and the agent selector is extended with two new quantors. In Section 6.2 the programming paradigm is demonstrated by rewriting the example Coffee Domain agents to use the new mental model semantics. Finally, the effect of using mental models on the programming effort is evaluated in the concluding section.

6.2 Programming with mailbox semantics

Let's take another look at the programs of the agents from the Coffee Domain. Figure 6.1 lists the `perceptrules` section of the `maker` agent.

We see that apart from those on lines 16-21, all percept rules query the mailbox to see if an agent has stated anything about its beliefs or goals. The general pattern can be described as follows:

"If the mailbox contains a received message from `S`, process its content according to its mood, then remove it from the mailbox."

There are several issues with this pattern:

1. To prevent repeatedly executing the same rule after a message has been received, either the `received` fact has to be deleted from the mailbox, or it has to be tested whether the intended action was already performed.

   The first solution requires an explicit extra action in the rule. Another disadvantage is that it prevents other rules from acting on this received message.

   The second solution, testing if the action was already performed, involves testing if the state that would be achieved by the action is already achieved. Testing this is not always directly possible, for example in the case of an action on the environment that does not have a perceivable effect.

GOAL

```
 1 perceptrules {
 2     % capability exploration:
 3
 4     % ask each agent what they can make
 5     if bel(agent(A), not(me(A)), not(canMake(A, _))) then sendonce(A, ?canMake(A, _)).
 6     % answer any question about what this agent can make
 7     if bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prod))
 8         then sendonce(A, :canMake(Me, Prod)).
 9     % process answers from other agents
10     if bel(received(Sender, canMake(Sender, Products)))
11         then insert(canMake(Sender, Products))
12         + delete(received(Sender, canMake(Sender, Products))).
13
14     % update beliefs with those of others (believe what they believe)
15     if bel(agent(A), received(A, have(X))), not(bel(have(X))) then insert(have(X)).
16
17     % If we need some ingredient, see if we can make it ourselves
18     if goal(have(P)), bel(requiredFor(P, R), not(have(R))),
19         bel(canMakeIt(Me, R), me(Me)) then adopt(have(R)).
20     % else try to find a maker for it
21     if goal(have(P)), bel(requiredFor(P, R), not(have(R))),
22         bel(canMakeIt(Maker, R), not(me(Maker))) then sendonce(Maker, !have(R)).
23
24     % if some machine seems to need a product, tell it we have it
25     if bel(agent(Machine), received(Machine, imp(have(X))), have(X))
26         then sendonce(Machine, :have(X)).
27 }
```

Figure 6.1: Coffee maker agent's percept rules

2. The `sent` and `received` facts in the mailbox contain no temporal information. This means that it is possible that an incoming message contradicts an earlier received message. For example, suppose an agent receives from some agent the message '`:`φ', and then at a later time receives from that same agent the message '`!`φ'. One could conclude that the sending agent no longer believes φ. But both messages are present in the mailbox, so a rule that checks for indicative messages will still find the first message, which represents an outdated state.

   Again, a solution is to make sure that at the end of an agent's run cycle, all `received` facts are processed and deleted from the mailbox.

3. The extra actions needed to handle these situations lead to redundant looking code. See for example the rule on lines 10-12 in Figure 6.1.

These issues show that the programmer has to make choices regarding how to keep the view on the other agents consistent, and how to make sure that rules handle incoming messages gracefully, i.e. do not infinitely repeat. The mailbox semantics, while flexible, force the programmer towards imperative style programming to handle these issues. This is undesirable, because the GOAL language promotes a declarative style of programming. The goals and beliefs of an agent are specified declaratively, and the mental state conditions in the percept- and action rules are specified by belief- and goal operators. To maintain this paradigm, mental models are introduced.

Mental models are models of the mental states of other agents that an agent knows of, and are automatically updated using incoming communication. The agent programmer can query these mental models in a similar way to querying the agent's own mental state. Because mental models are automatically updated, the agent programmer no longer needs to constantly check the consistency of the information it keeps on the beliefs and goals of other agents. The syntax of a mental model query is intuitive and closely resembles that of a normal mental state query, which improves readability of the GOAL code.

6.3 Models of mental states

A mental model is is a model of a mental state:

$$\mathcal{M}_a = \langle \Sigma_a, \Gamma_a \rangle$$

where a is the name of the agent whose mental state is modeled, Σ is a belief base and Γ is a goal base. Notice that, contrary to an agent's own mental state, no message bases, percept bases, or any other kind of database are present. The mental model of an agent models beliefs and goals.

6.3.1 Initialization

Every agent maintains a mapping of agent names to mental models. If an new agent becomes known to an existing agent, a mental model is initialized and added to the mapping. This can happen when the agent platform informs the existing agent of the launch (birth) of the new agent, if the agent platform is configured to do so. In any case the existing agent will initialize a mental model when it receives a message from an agent whom it doesn't know yet.

6.3.2 Querying

The objective of maintaining models of the mental states of other agents is to be able to reason about those mental states, i.e. about the beliefs and goals of those agents. An agent would like to be able to query the mental state of an agent, but since this is not possible directly ([19]), the *model* of the mental state is queried instead. For the sake of intuitive programming, the programmer's interface for querying mental models closely resembles that for querying the agent's own mental state.

The format of a mental model query literal is:

$$<\!agentselector\!>.<\!queryoperator\!>\ (<\!propositional\ content\!>)$$

Here, the *queryoperator* is one of `bel`, `a-goal`, `goal` or `goal-a`, and *propositional content* is the actual query. So far this is the same as for querying the agent's own mental state. The difference lies in the *agentselector*. This agentselector specifies the agent(s) whose mental model is to be queried. It is an extension of the agent selector used for specifying the recipients of a message, as described in Section 5.6.1. It's syntax is the same as given in Figure 5.6.

The examples given in Figure 6.2 illustrate some uses of the agent selector in querying mental models. Recall from Section 5.6.1 that upon evaluation of a `send` action, the agent selector is resolved to a list of agent names, using the agent base. Similarly, upon evaluation of the mental state condition

GOAL

```
1 if agent2.goal(have(beans)), bel(have(beans)) then giveTo(agent2, beans).
2
3 if bel(agent(A)), A.bel(canMake(A, Products)), not(bel(canMake(A, _)))
4     then insert(canMake(A, Prods)).
5
6 if allother.bel(shapeOfWorld(flat)) then insert(shapeOfWorld(flat)).
```

Figure 6.2: Example mental model queries

of an action rule, the agent selector of each mental literal is resolved. The mental literal is then queried on the mental model of each of the resulting agents. Variable bindings are passed along each query of a mental model, in the same way as happens with individual mental literals. In fact, the effect of resolving an agent selector can be seen as 'expanding' the agent selector into a conjunction of mental model queries of single agents. See Figure 6.3.

GOAL

```
beliefs {
  % Given that the agent knows the following agents:
  agent(alice). agent(bob). agent(charlie).
  me(charlie).
}
program {
  % The following query:
  if allother.bel(shapeOfWorld(S)) then insert(shapeOfWorld(S)).

  % Will be resolved as
  if [alice, bob].bel(shapeOfWorld(S)) then insert(shapeOfWorld(S)).

  % Which will expanded to
  if alice.bel(shapeOfWorld(S)), bob.bel(shapeOfWorld(S))
    then insert(shapeOfWorld(S)).
}
```

Figure 6.3: Effect of resolving agent selector in a mental model query

If (`charlie` believes that) `alice` believes that `shapeOfWorld(flat)`, then after querying the mental model of `alice` the substitution set `[[S/flat]]` is passed to the next mental literal, in which each substitution is applied to the database formula before the query is performed. In the example this means that the query `shapeOfWorld(flat)` is performed on the mental model of `bob`. If `bob` does not believe `shapeOfWorld(flat)`, the mental state condition fails.

Variables

Just as with agent selectors in `send` actions, variable-type agent expressions must be bound at the time of evaluation. This is the reason that the mental literal `A.bel(..)` on line 3 in Figure 6.2 is preceded by `bel(agent(A))`, to get a binding of every agent name to `A`.

Extra quantors: `some` and `someother`

In addition to the quantors that are used in agent selectors for the `send` action, `self`, `all` and `allother`, two additional quantors are allowed for use in mental model queries;

- **`some`**: if *any* mental model yields a result to the query, that result is returned.
- **`someother`**: if a mental model of *any other* agent yields a result to the query, that result is returned.

This can be used when we want to check if *some*(*other*) agent believes something or has some goal. Just like `all` and `allother`, these quantors may not be used in combination with other agent expressions in one agent selector. This is because agent selectors such as `[some, all]` or `[X, someother]` are ambiguous and confusing.

6.3.3 Updating

A mental model represents the beliefs and goals of another agent. Mental models are updated automatically by means of conversational implicatures (see 3.2) based on the mood and content of an incoming message. By analyzing such an incoming message, an updated state of the modeled mental state of the sending agent can be deduced.

For example, if agent `maker` receives the message

```
!have(beans).
```

from agent `grinder`, `maker` may conclude that `grinder` has a goal to `have(beans)`, and thus update its mental model of `grinder` with this goal. But the fact that `grinder` has a goal to `have(beans)` implies that it does not believe `have(beans)`, else the goal would already have been achieved, according to its beliefs.[1]

In general, suppose an agent R receives a message $\langle \mu, \varphi \rangle$ from sending agent S. Here, μ is the mood and φ is the propositional content. The mental model of S $\mathcal{M}_S$ is updated to $\mathcal{M}'_S$ as follows:

- if μ is INDICATIVE: $\mathcal{M}'_S = \langle \Sigma_S \oplus \varphi, \Gamma_S \ominus \varphi \rangle$
- if μ is INTERROGATIVE: $\mathcal{M}'_S = \langle \Sigma_S \ominus \varphi, \Gamma_S \rangle$
- if μ is INDICATIVE: $\mathcal{M}'_S = \langle \Sigma_S \ominus \varphi, \Gamma_S \oplus \varphi \rangle$

After every update of a belief base, the goal base of that mental model is updated so that it stays consistent with the belief base. This process of updating the mental model is described in more detail in Section 3.6.

[1] Actual facts or the receiving agent's own beliefs are irrelevant here; the purpose of the mental model is *modeling* the mental state of an agent.

6.3.4 Mental state as a mental model

The agent selector for mental model queries allows the use of the `self` quantor. In the `send` action, the semantics are that the message is sent to the sender itself. In mental model querying it means that the own mental state (which is not a *model*) is queried. In fact, all mental literals without any agent selector get an implicit `self` set as agent selector. So

```
if bel(weather(sunny)) then goTo(beach).
```

is interpreted as

```
if self.bel(weather(sunny)) then goTo(beach).
```

6.4 Using mental models in GOAL programs

Taking the example of the Coffee Domain from Chapter 5, we can revise the programs to make use of mental models.

Capability exploration To find out what the other agents can make, the following percept rules are used in the program:

```
% ask each agent what they can make
if bel(agent(A), not(canMake(A, _)) then sendonce(A, ?canMake(A, _)).

% answer any question about what this agent can make
if bel(me(Me), received(A, int(canMake(Me, _)), canMake(Me, Prods))
  then send(A, :canMake(Me, Prods)) + delete(received(A, int(canMake(Me, _)))).

% process answers from other agents
if bel(received(Sender, canMake(Sender, Products)))
  then insert(canMake(Sender, Products)) + delete(received(Sender, canMake(Sender, Products))).
```

The first rule can inspect the mental model of the variable agent `A` to see if it (believes it) can make anything:

```
% ask each agent what they can make
if bel(agent(A)), not(A.bel(canMake(A,_))) then sendonce(A, ?canMake(A,_)).
```

Depending on the rest of the program, we can now opt to not perform the third rule of the above snippet. I.e. instead of believing what other agents believe, simply look at what they believe. This would also be possible with mailbox semantics, but the code would get rather unmaintainable. Also, it would require that the programmer makes a consistent policy on how to maintain the mailbox.

The second rule can use a mental model query to check if an agent does not know what this agent can make, and thus send this information to that agent:

```
% answer any question about what this agent can make
if bel(me(Me), canMake(Me, Prods), agent(A)), A.bel(not(canMake(Me, _)))
```

The third action rule can be replaced with

```
if bel(agent(A)), A.bel(canMake(A, Products)), not(bel(canMake(A, _))) then
    insert(canMake(A, Products)).
```

Alternatively, this action rule can be completely removed and

```
bel(agent(A)), A.bel(canMake(A, Prods)), bel(member(grounds, Prods))
```

can be used wherever the agent wants to test which agent can make `grounds`, for example.

Production delegation For production delegation, the `grinder` agent has the following action rule:

```
% if some agent needs grounds, then adopt the goal to make it
if bel(received(_, imp(have(grounds)))) then adopt(have(grounds))
 + delete(received(_, imp(have(grounds)))).
```

This rule can be replaced with one that uses mental models:

```
% if some agent needs grounds, then adopt the goal to make it
if someother.goal(have(grounds)) then adopt(have(grounds)).
```

Here we see the use of the `someother` quantor. If any other agent has a goal to `have(grounds)`, adopt that same goal.

Status updates When one agent has a product that it knows another agent needs, it tells that agent that it has the product:

```
% if some machine seems to need a product, tell it we have it
if bel(agent(Machine), received(Machine, imp(have(X))),
  have(X)) then sendonce(Machine, :have(X)).
```

The `received(..)` part of the mental state condition can be replaced with a mental model query:

```
% if some machine seems to need a product, tell it we have it
if bel(agent(Machine)), Machine.goal(have(X)), bel(have(X))
  then sendonce(Machine, :have(X)).
```

Here the rule reads more intuitively; "if a machine has a goal to have X and I have X then tell that machine that I have X".

On the receiving side, this information was processed by a rule that copied the beliefs from any received indicative message:

```
% update beliefs with those of others (believe what they believe)
if bel(agent(A), received(A, have(X))), not(bel(have(X))) then insert(have(X)).
```

This is a perfect example of how mental model queries can improve a program. In the mailbox semantics situation, the indicative message is manually processed and the fact is inserted in the agent's own belief base. At that point, the source of the statement (that `have(X)`) is lost. Using mental models, we can completely remove the above action rule, because the fact is already automatically reflected in the mental model of the sending agent. Thus, wherever a belief query is made to see if this agent believes `have(X)` (`bel(have(X))`), that can be replaced with a query that checks if *any* agent believes `have(X)`: `some.bel(have(X))`.

Consistency So far we have replaced checking the mailbox by querying mental models. This already improves readability, but there is another important advantage of using mental model queries to guess an agent's mental state. Consistency between goals and beliefs, and updates thereof are not automatically done for the mailbox contents. Mental models however, always reflect the latest consistent mental state, as far as the receiving agent could possibly know by observing communication.

This means that in a situation where an agent sends the message `:have(beans)`, indicating that (it believes) it has `beans`, and later sends the message `!have(beans)`, because the `beans` have run out and are needed again, the mental model of the sender is automatically updated by adding the goal to `have(beans)` to the mental model's goal base and removing that fact from the mental model's belief base. Because the order of reception of messages is not maintained in the mailbox, it is somewhat difficult to determine what the current mental state of the sender is. When using only the mailbox semantics, every incoming message should be checked, its information processed and deleted every round to maintain a consistent view of the sending agent's mental state. The mental model of the sending agent on the other hand is automatically updated. After processing the second incoming message, the fact `have(beans)` is retracted from the belief base and added to the goal base.

6.5 Conclusion

In this chapter the agents from the example multi-agent system have been rewritten by replacing queries on the mailbox by mental model queries. In one occasion, a rule could be completely removed, because its function is already implemented in the mental models.

The shift from mailbox semantics to mental models results in code that has a more declarative, goal- and belief-oriented nature compared to the message-oriented mailbox semantics.

Mental models are automatically kept consistent with the received communication, and therefore provide a robust way of maintaining a model of another agent's mental state. When using only the mailbox semantics, a programmer would have to ensure this consistent view of another agent's beliefs and goals through complicated program rules.

The use of mental model queries can significantly improve readability of programs and makes it much easier to reason about the beliefs and goals of other agents. In Chapter 5 we have reasoned about other agent's mental states by inspecting the mailbox contents, and manually processing the messages. Mental models provide a consistent and opaque programmer's interface to those operations. It liberates the programmer from having to deduce facts about the beliefs and goals of others, because these deductions are done automatically and are accessible through the mental model queries.

Chapter 7

Distributed Multi-Agent Systems

7.1 Introduction

In a distributed agent world, on the computational level, agents can run on different hosts. More than one agent can run on a single host of course. The hosts may vary in their physical location, or they may be heterogeneous in their hardware configuration (e.g. embedded system vs. supercomputer). The reason for distributing agents across different hosts depends on the application, but are usually related to:

1. **load-balancing**. When agents perform or are responsible for heavy computational work, distributing them across hosts can benefit from parallelism. In some cases, the agents will be the per-host managers of the operation, and handle the communication of data and results with some aggregator.

2. **local operations**. In some situations, agents can operate the specific device they run on. This is the case in robotics, or in embedded agents. In a real-life realization of the Coffee Domain example introduced in Chapter 5, a sophisticated, integrated coffee maker could be controlled by the coffee maker agent and the coffee grinder agent.

On the conceptual (agent) level, agents usually do not need to know that they run on different hosts. The agent programmer should not need to be concerned about how the agents are distributed and if they end up communicating across hosts or not. In other words, the distribution of agents is not a matter of agent programming.

For the GOAL platform, such distribution of agents should also be possible. The objective is to provide a way for agents to communicate with other agents that may run on a different GOAL platform, on a different host. The fact that the agents are distributed should have the least possible impact on the agent's programs. Preferably, the agent programs do not need to be changed at all to move from a non-distributed setup to a distributed setup.

In this chapter the options for implementing this functionality are explored. First the existing technologies in the field are surveyed. Then an implementation is proposed and implemented. Finally, the implementation is evaluated.

7.2 Requirements of a distributed agent system middleware

For the GOAL system, the main goal is to allow agents to transmit a message to another agent. There is no need for conversation tracking, mobility (the ability for agents to move from one host to another), or integrated remote debugging. An implementation of a middleware should not introduce too much complexity into the GOAL platform. Therefore, we aim for a minimal implementation of a middleware that satisfies the following criteria:

1. When an agent sends a message, the message should appear in the receiver's mailbox.

2. Agents can address each other in communication using the names from the `agent()` facts.

3. The effort to configure a GOAL platform to be part of a distributed MAS should be minimal.

4. The effort of launching the agents of a MAS distributed across multiple hosts or GOAL platforms should be low.

From these criteria sub-criteria follow, as we will see in the remainder of this chapter.

7.3 Existing technologies

In the field of agent programming, many technologies exist that provide some sort of interface between the agents and the communication layer. The purpose is generally to abstract all or most of the inherit complexities that come with distributed computing.

7.3.1 Agent middlewares

In this subsection we explore agent frameworks such as JADE[1] and AgentScape[3]. These frameworks aim to provide a means to build a distributed multi-agent system.

JADE JADE tries to simplify development of FIPA[7] compliant agents by providing a set of system services and agents. It hides from the agent builder the aspects that are not a part of the agents or the agent application, such as message transport, encoding and parsing of the messages and agent life-cycles. See Section 2.3.3 for a detailed description of the JADE framework. The JADE framework provides a set of service agents, that are specified by the FIPA specifications. These agents are a Directory Facilitator (DF), an Agent Management System (AMS) and an Agent Communication Channel (ACC). These agents are used in controlling the life-cycle of agents (AMS), registration of and searching for agent services (DF) and communication between agents (ACC). The ACC handles all communication between agents, and transparently selects the most efficient transport mechanism; communication between agents on the same platform is done using event signaling and direct passing of Java objects, while communication between agents on different platforms relies on Java RMI (see next subsection). Agents each run in an own Java thread, but allows agent programmers to implement multi-threaded behaviour through the implementation of `Behaviours` which are scheduled cooperatively per agent.

JADE specifically aims at FIPA compliance and interoperability of agents.

AgentScape AgentScape is a middleware layer that supports large-scale agent systems. It has a ‘less is more’ philosophy, in that it attempts to provide only those mechanisms that are required to operate a large-scale agent system, while preserving the freedom for agent developers to implement their own agent model. AgentScape does not impose a specific agent model, unlike JADE. It uses XML-RPC for communication between agents and interaction with a *LookupService*, a lookup service. AgentScape agents are started in their own Java thread to allow concurrent execution.

7.3.2 Java RMI

Java’s Remote Method Invocation (RMI) is not a middleware, but a technology that is a feature of Java. It allows objects to invoke methods on other objects that run on a different JVM (that may run on a different host). When such a call is made, the arguments of the method are marshalled at the caller’s end, sent over the network and unmarshalled at the callee’s end. The return value of the method is again marshalled, sent and unmarshalled. All argument types and return value types should be serializable.

RMI can be seen as a kind of client-server model implementation. Clients call methods on a server, which is a remote object. Remote objects can register themselves in a registry using a label. Client objects can then look up this label in the registry to obtain a reference to the remote object. Any host that has remote objects should run an RMI registry, as it is not possible, for security reasons, to bind an object in a registry that is not on the local host. So a client should at least know on which host the remote object lives.

The methods that can be called on a remote object are specified in an interface that extends from `java.rmi.Remote`. For example, a time service can be specified with this interface:

```
public interface TimeService extends java.rmi.Remote {
    public java.util.Date getTime() throws java.rmi.RemoteException;
}
```

An implementation of the `TimeService` should extend `UnicastRemoteObject` and of course implement the `TimeService` interface.

To invoke a method on a remote object, a client looks up the remote object in the remote registry. Instead of getting the complete remote object, a so called *stub* is returned. This stub has the same interface as the remote object, and handles the call such that it is marshalled and sent to the remote object, and returns the result to the caller.

Java RMI as middleware

RMI provides a mechanism for objects on different JVMs or hosts to communicate with each other on the object level. When we regard the agents of a GOAL platform as remote objects, communication of messages between agents could be realized by specifying and implementing a simple interface like the following: When an agent wants to send a message to another agent, it can look up that agent by

Java
```
import java.rmi.Remote;
import goal.middleware.Message;

public interface AgentInterface extends Remote {
    // Accepts a message and puts it in the agent’s message in queue.
    public void acceptMessage(Message m) throws RemoteException;
}
```

Figure 7.1: Interface for accepting messages

its agent name in the registry and call its `acceptMessage` method.

7.3.3 TCP/IP Sockets

Both Java RMI and the XML-RPC implementation of AgentScape rely on TCP/IP sockets for communication across JVMs or hosts.

For communication between threads on different JVMs or hosts, falling back to a low level implementation is always an option. In that case, an implementation needs to take care of the opening and closing of sockets, buffered input- and output streams, management of open/available ports and continuous listening for new incoming connections.

7.3.4 Conclusion

While agent frameworks such as JADE and AgentScape provide a full-featured environment for distributing agents and having them communicate, they usually impose a certain agent model (JADE's *behaviours*) or execution model (JADE and AgentScape run one thread per agent). This is restrictive when attempting to integrate the framework within the GOAL interpreter, and adds to its complexity.

To provide the minimal yet sufficient functionality for communication of agents in a distributed setup, a lightweight implementation using Java RMI is chosen.

7.4 Design of the GOAL middleware

For GOAL an approach is chosen where a distributed MAS can consist of agents which are launched on different GOAL platform. A GOAL platform is one instance of the GOAL interpreter, with usually the GOAL IDE GUI, but optionally only running the stand-alone version. In this text it is assumed the user runs the GUI. The user can launch one MAS file at a time per GOAL platform. Per GOAL platform the user sets the name of the host that runs the central GOAL platform. This central platform is a special one in that it maintains the central registry of agents in the whole distributed MAS. In the case of multiple GOAL platforms on that host, the first one to launch a MAS will be designate itself as central platform, and subsequent platforms will automatically detect the existing registry.

A simple event-based updating mechanism ensures that all agents remain up-to-date with respect to the list of agents in the global MAS. When a user of a GOAL platform launches a MAS file, all agents in that MAS file are launched according to the launch policy (see Section 5.2.2). These agents are then registered and announced in the global MAS. That means that all existing agents become aware of the newly launched agents. All newly launched agents are also made aware of the existing agents. The agents launched on one platform are not visible or controllable in another platform, but from the perspective of an agent there are no platform or host boundaries; to the agents it looks as if they all live on the same platform. Whenever the user kills an agent or a (local) MAS, all other agents are notified of the death of these agents. Those remaining agents remove the existence of the deceased agents from their beliefs and remove corresponding mental models.

Agents communicate by obtaining a reference to the receiving agent from the central registry, and sending the message to that agent. The message is buffered at the receiving agent in a message queue until the beginning of that agent's next run cycle. All messages in the message in queue are then processed and placed in the agent's mailbox and the mental models of the respective sending agents are updated. From this point onwards the agent can reason about the communication, either through the mailbox or the mental models.

Environments GOAL offers support for connecting to an environment. An environment can spawn entities for which, according to the launch policy specified in the MAS file, GOAL agents are launched. Communication between agents is then no different from the situation without an environment. However, there are also percepts and actions which are to be communicated between the environment and

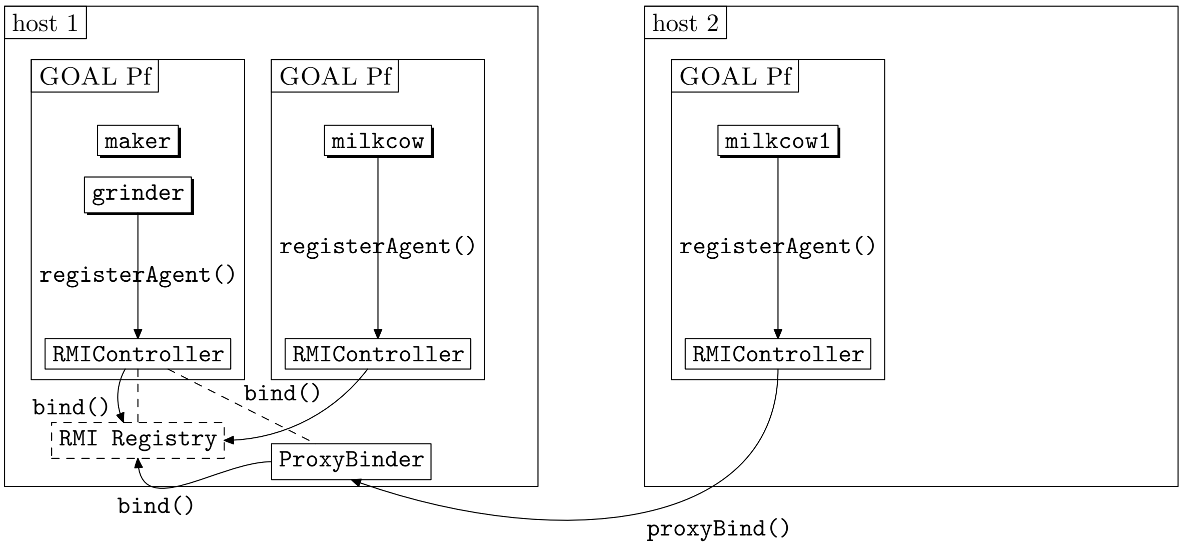

Figure 7.2: Three GOAL platforms on two hosts

the agents. Environments and implementing support for distributed participation of agents in an environment is outside the scope of this thesis. When implementing this support, one has to determine at runtime which GOAL platform should launch an agent, since multiple GOAL platforms may have launch policies that could launch that agent.

7.5 Implementing the GOAL Middleware

Inter-agent communication is implemented by means of Remote Method Invocation where the sending agent's Java object invokes a method on the receiving agent's Java object, passing the `Message` object as a method parameter. In terms of Java objects, the sender may be viewed as the client and the receiver as the server. To be able to send and receive messages, the `Agent` class implements the interface as listed in Figure 7.1. But first, the sending agent must obtain a reference to the remote object implementing the receiving agent.

7.5.1 The `RMIController`

All GOAL platforms perform their RMI-related operations via an `RMIController` object. This object provides an interface between the GOAL platform and the RMI operations. It transparently handles binding, unbinding, lookup and listing of agents, regardless of whether the central RMI registry runs on the local host or on a remote host. It is initialized with the host name or IP address of the host that runs the GOAL platform that has initialized the RMI registry. If the host name or IP address refers to the local host, the `RMIController` checks if an RMI registry already exists on the local host. If not, it will launch it, together with the `ProxyBinder` service that is discussed in Section 7.5.2.

7.5.2 Registration and lookup of agents

Since agents in a distributed MAS can be launched on different GOAL platforms, and agents themselves are not aware of this distribution, some mechanism must be made available that allows a sending agent to obtain the reference to the receiving agent in order to call its `acceptMessage` method. The RMI Registry binds names to remote objects. A reference to a remote object (a stub) can then be obtained by looking up the name of the object in the registry. Objects can only be registered in a registry on the same host it runs on, so to get a reference to this object, the registry of the host that object runs on must be contacted to perform a lookup of the object's name. This implies that the client must know on which host the server is located. A sort of centralized locating service is therefore required. It would be most efficient if objects could be registered in one central RMI registry, and all clients could find all servers in that single known RMI registry. Every GOAL platform would then only need to know the hostname (and port) of the central registry.

Due to security restrictions of RMI, binding and unbinding an object in a remote registry is not allowed, so the above solution is not applicable. To solve this problem, binding in the remote registry is done indirectly via a `ProxyBinder` object. This `ProxyBinder` is a remote object that has a commonly known name, so all `RMIControllers` can look it up. Exactly one instance of this object lives on the same host as the registry and is bound in it to that commonly known name. It implements the `proxyBind` and `proxyUnbind` methods, which call the respective method on the local registry, allowing objects to bind and unbind themselves with a name in that registry by proxy. The `ProxyBinder` then registers the server object in the registry. This *is* allowed, because the `ProxyBinder` resides on the same host as the registry. This process is illustrated in Figure 7.2.

When an agent is launched, its GOAL platform checks whether the central RMI registry resides on the local host. If so, it uses the standard RMI methods to bind the agent object in the registry. If the RMI registry resides on some other host, a lookup is performed to obtain a reference to the `ProxyBinder`. The agent and its name are then passed to the `proxyBind` method of the `ProxyBinder`. The `ProxyBinder` then binds the agent to its name in the registry. Unbinds are performed in a similar way.

Lookups can always be performed directly on the registry and do not require the aid of a proxy like the `ProxyBinder`.

It should be noted that whenever a remote object (i.e., an object that extends `UnicastRemoteObject`) is passed as an argument — or returned as a return value of a remote method invocation, the whole object is not marshalled, but rather the stub belonging to that object. This considerably reduces communication payload when passing these remote objects. When for example a GOAL platform's `RMIController` registers an agent using the `proxyBind` method of the (remote) `ProxyBinder` it calls `proxyBind(agent.getName(), agent)`. The RMI system automatically replaces the `Agent` object with an instance of `Agent_Stub`, which holds all information to reference that `Agent` object. Whenever in this text is spoken of 'passing an agent to a remote object', it is assumed that RMI handles this.

7.5.3 Agent/MAS lifecycles

In a dynamic distributed multi-agent system, the agents in the system may have been launched on different GOAL platforms. Suppose we want to extend the Coffee Domain with the production of lattes. The coffee maker agent can produce lattes, which requires coffee (which it can make itself), and milk. Milk is produced by milk cows. Previously, a milk cow was added to the MAS file. Now we take a more distributed approach. Suppose there is not just one cow, but many, and they live not near the coffee maker and coffee grinder, but on a dairy farm. The dairy farm is launched on a different GOAL platform that runs on a different host from that of the coffee machines. There is no way to launch a MAS — or individual agents — on a remote GOAL platform from another GOAL platform. A MAS can only be launched locally, i.e. by selecting a MAS file in the GOAL IDE of the platform the MAS should be launched on. In order to distribute the agents over two separate GOAL platforms, two separate MAS files thus need to be constructed.

```
Coffee machines
agentfiles {
  "coffeemaker.goal".
  "coffeegrinder.goal".
}
launchpolicy {
  launch maker:coffeemaker.
  launch grinder:coffeegrinder.
}
```

Figure 7.3: Mas file for the coffee machines

```
Dairy farm
agentfiles {
  "milkcow.goal".
}
launchpolicy {
  launch milkcow[5]:milkcow.
}
```

Figure 7.4: Mas file for the dairy farm

Let's consider the example of the Coffee Domain, where the dairy farm is a system of `milkcow` agents which will produce milk for the coffee machines. The dairy farm gets a separate MAS file from the `maker` and `grinder` agents. The MAS file of the dairy farm launches 5 agents of the type `milkcow`. Suppose now that the coffee machine MAS file is first launched on a GOAL platform that is set up to start the RMI registry. Once the agents are launched, they are aware of each other's existence through the `agent(..)` facts that are inserted in their respective belief bases upon launch, as discussed in Section 5.6.2. Of course, these agents are not aware of any `milkcow` agents, because they do not yet exist.

Now when another GOAL platform that is set up to connect to the host of the abovementioned platform launches the dairy farm MAS file, five `milkcow` agents are launched. These agents are also bound in the central RMI registry, by the process described in Section 7.5.2. This binding in the registry does not make all agents in the MAS aware of the new agents or vice versa. It is technically possible to implement a pull model in which each agent gets the list of agents from the registry via the `RMIController`'s `getAllAgentNames` at the beginning of its run cycle, and then update its own agent base accordingly. However, this is rather expensive, since it requires communication with the registry every round of every agent, while changes in the agent list occur not so often compared to the run cycle frequency. Instead a push model is chosen, where changes in the agent list (births and deaths) are propagated to all agents. Also, when an agent is launched, the list of existing agents is obtained from the registry via the `RMIController` and used to insert `agent(..)` facts into the new agent's belief base. To allow a GOAL platform to inform a remote agent that a new agent has been born or has died, the interface from Figure 7.1 is extended as shown in Figure 7.5.

```java
import java.rmi.Remote;
import goal.middleware.Message;

public interface AgentInterface extends Remote {
    // Accepts a message and puts it in the agent's message in queue.
    public void acceptMessage(Message m) throws RemoteException;

    // Adds the agent to the agent base and constructs a mental model.
    public void handleAgentBirth(String agentName) throws RemoteException;

    // Removes the agent from the agent base and removes the mental model.
    public void handleAgentDeath(String agentName) throws RemoteException;
}
```

Figure 7.5: Agent interface extended with birth- and death handling methods

Now, when each agent of the dairy farm is launched, its GOAL platform notifies each agent already present in the MAS (the `maker`, `grinder` and any already thusfar launched `mikcows`) of the birth of this new agent by calling the `handleAgentBirth` method on those existing agents. After launch of all five `milkcows`, agent `maker`'s belief base will look like Figure 7.6. When an agent or a complete GOAL

```
beliefs {
  agent(maker).
  me(maker).
  agent(grinder).
  agent(milkcow).
  agent(milkcow1).
  agent(milkcow2).
  agent(milkcow3).
  agent(milkcow4).

  ... % other beliefs
}
```

Figure 7.6: Agent `maker`'s belief base after dairy farm is launched

platform is killed, the remaining agents are notified in an analogous way. If for example the dairy farm is closed, all `milkcow` agents are first killed. For every `milkcow`, all remaining agents have their `handleAgentDeath(milkcowname)` method invoked. This results in that the `agent(milkcow..)`

facts are deleted from the belief base, and the corresponding mental model removed. This way, an agent knows that if `agent(A)` is the case, then that agent `A` exists and can be communicated with.

Name clashes The possibility exists that a second instance of the dairy farm is launched. This would cause the GOAL platform to attempt to register those new `milkcows` with the registry. When the GOAL platform selects the names `milkcow`, `milkcow1`, ..., `milkcow4`, this results in a name clash with the existing `milkcows`.

To prevent name clashes from occurring, the GOAL platform's naming mechanism, as described in Section 5.2.2, is adapted to take into account the names of all existing agents in the MAS when constructing unique names. This means that if there are already five `milkcows` in the MAS, launching a second instance of the dairy farm will yield five agents with names `milkcow5` till `milkcow9`.

To accomplish this, every time the GOAL platform launches an agent, it retrieves the list of existing agents (globally) from the `RMIController`, which in turn gets it directly from the RMI registry.[1] A unique name is then determined for the new agent. Figure 7.7 illustrates this process.

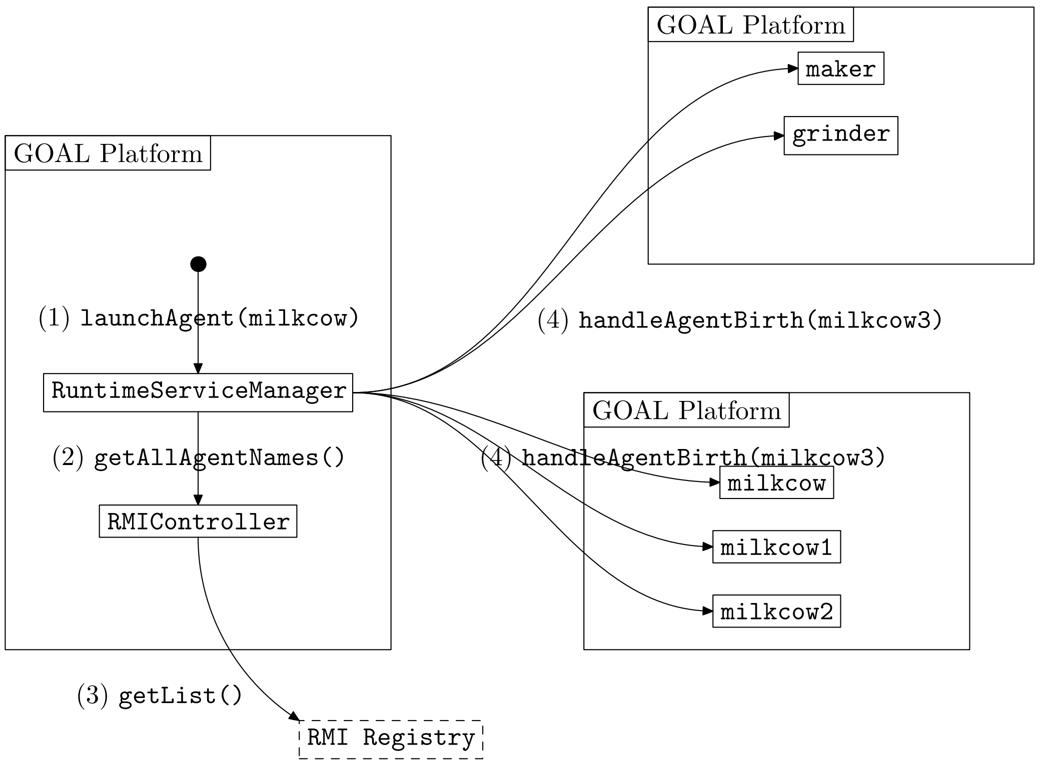

Figure 7.7: Launch and announcement of a new agent. First, the GOAL platform launches the new agent (1). The `RuntimeServiceManager` retrieves the list of agent names (2),(3). It then determines the next globally unique name (`milkcow3`) and announces this agent name to the existing agents (4).

[1]The `RMIController` removes names of objects in the RMI registry that are not agents, like '`ProxyBinder`', from this list.

Relaunching agents Another issue arises when the dairy farm is launched again after closing it. Announcements of the new `milkcows` are done as normal and after the `milkcows` are all launched, agent `maker`'s belief base again looks like Figure 7.6. The problem is that the mailboxes contain facts about sent and received messages relating to the *old* `milkcows`. So an action rule that once asks an agent what it `canMake` for example, using the `sendonce` action, will not do so again for the new agent with the same name, even though that agent may have different capabilities. Similarly, the agent that answers such a question using the program rule from Figure 7.8 will send that answer only once.

GOAL
```
if bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prods))
    then sendonce(A, :canMake(Me, Prods)).
```

Figure 7.8: Action rule answering capability questions

If the agent that posed the question gets killed and relaunched with the same name, it will not know the answer (any more), and per its program it will pose the question again. However, the receiving agent will not notice any change in the mailbox, because the fact `received(sender, int(canMake(Me, _)))` that gets inserted in the mailbox was already there. The answer was already sent, so the `sendonce` action will not send the indicative message again. This is not the desired behaviour, because the newly launched agent will not get the requested information this way.

One possibility is to not allow the reuse of agent names, so names are unique over the entire life span of the MAS. But this also prevents the relaunch of special agents that have a well-known name.

Instead an approach is chosen in which the programmer can take these situations into account by altering the mailbox handling action rules. In the example from Figure 7.8, if we want this agent to answer all subsequent questions from the same agent(name), we can replace `sendonce` with `send`, and remove the `received(..)` fact from the mailbox after sending the response. This is listed in Figure 7.9.

GOAL
```
if bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prods))
    then send(A, :canMake(Me, Prods)) + delete(received(A, int(canMake(Me, _)))).
```

Figure 7.9: Reusable action rule version of 7.8

A more challenging issue is found in another action rule of the `milkcow`, listed in Figure 7.10. This universal rule checks if an agent has a goal to `have` a product, and this agent believes it has that

GOAL
```
% if some machine seems to need a product, tell it we have it
if bel(agent(Machine), not(me(Machine))), Machine.goal(have(X)),
    bel(have(X)) then sendonce(Machine, :have(X)).
```

Figure 7.10: Universal product-supplying action rule

product, it informs the former agent of that fact. The problem is illustrated by the following sequence of events:

1. agent `maker` has a goal to `have(milk)`
2. agent `maker` sends `!have(milk)` to `milkcow`, which then believes `maker.goal(milk)`.
3. the `milkcow` sends `:have(milk)` to `maker`, *once*.
4. the `maker` is relaunched. `milkcow` deletes its mental model of `maker` and initializes a new one for the relaunched `maker`.
5. steps 1 and 2 are taken again.
6. the mental state condition of the action rule in Figure 7.10 is satisfied, but `sent(maker, have(milk))` is also believed by `milkcow`, so the `sendonce` action is not executed again.

So the problem is that `milkcow` already acted on the fact that `maker` has a goal to `have(milk)`. The newly received imperative does not change anything for `milkcow`, which thinks it has already acted. What we want is that upon receiving a new imperative, the ‘lock’ on the `sendonce` is released. The fact preventing the `sendonce` from executing is that the agent already has `sent(...)`. The solution therefore is an action rule, or a percept rule, that deletes this `sent` fact from the mailbox when such an imperative is received. See Figure 7.11. The imperative itself should also be deleted from the mailbox

GOAL
```
if bel(received(A, imp(have(P))))
    then delete(sent(A, have(P))) + delete(received(A, imp(have(P)))).
```

Figure 7.11: Mailbox cleaning rule

to prevent repeated execution of this rule, leading to repeated execution of the `sendonce` action.

7.5.4 Synchronization

In every (distributed) system where concurrent processes interact there is the issue of synchronizing those interactions. In GOAL, the interaction is performed through the passing of messages. Since agents in different GOAL platforms run concurrently, this may present problems when one agent tries to deliver a message in another agent while that receiving agent is performing operations on its mental state at the same time.

To minimize concurrent access to an agent’s resources, messages are not directly inserted into the mailbox upon delivery, but are buffered in a message in queue. At the beginning of the receiving agent’s run cycle, it processes all messages in this queue and performs the corresponding operations on its mailbox and respective mental models. All access to the message in queue is synchronized by means of a Java `SynchronizedList`, which synchronizes concurrent access to it.

7.6 Conclusion

In this chapter the requirements and criteria for a middleware implementation for the GOAL platform were investigated. To provide a minimal set of features, an implementation of middleware functionality was made, based on Java RMI. While restrictions on binding and unbinding agents in the RMI registry made the introduction of a `ProxyBinder` necessary, the implementation was relatively straightforward. The resulting RMI-based middleware is capable of handling the sending of messages across hosts, and the transparent registration and referencing of agents. Also, situations where agents are killed or relaunched were investigated and taken into account in the middleware implementation.

Investigations into the effect of distributing agents across GOAL platforms on the execution of the whole MAS led to the conclusion that in certain situations, a slight change of how messages are dealt with in the GOAL code is sometimes needed to deal with the relaunching of agents. These changes relate to the mailbox paradigm, and are intuitive when one thinks about the mailbox contents.

To investigate and demonstrate the impact of distribution on the GOAL programs, the Coffee Domain example was taken and distributed across two GOAL platforms. The concurrent execution of agents did not present any problems for the agents. A minor adaptation of the mailbox handling was necessary to account for relaunching of other agents, and then to continue to function for those newly relaunched agents, but all in all the increased burden on the programmer resulting from distribution was marginal. Also, the effort to setup and launch a MAS distributed amounted to splitting a MAS file into multiple MAS files, and launching each of those on a separate GOAL platform. Each of those platforms needed to be configured only with the name or IP address of the host on which the GOAL platform with the central RMI registry was started.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

In this thesis the agent programming language GOAL has been extended with communication. The aim was to provide pragmatic, usable communication constructs to the programmer while building on a well-founded theoretic base which would allow to define a formal semantics for the communication.

To get an idea of the work that was already done in the field and evaluate their use for this work, several state of the art agent technologies were reviewed. Two main concerns with respect to these technologies were identified: having a vast and ambiguous performative set, and lack of a formal semantics.

In this thesis I have followed the approach of Jason in reducing the performative set. The communication constructs that were introduced allow an agent to express statements about its beliefs and goals in one of three possible moods. These moods are the major moods as identified by Harnish, and inspired by Grice's work on conversational implicatures, were selected for their usefulness in communication between GOAL agents.

The semantics of the communication constructs respects the notion that agents cannot directly inspect each other's mental state. Therefore, the semantics does not refer to another agent's beliefs or goals. Instead, Grice's notion of conversational implicatures was taken as a theoretic basis to determine what *can* be inferred from incoming communication.

These inferred facts were used to construct mental models, which *model* the mental state of the other agent. These mental models are updated automatically upon receipt of a message, based on its mood and content. Programming constructs were added to the GOAL language to query the mental models. Using these mental model queries, agents can reason about the mental states of other agents. It is important to note here that the mental states of other agents are not queried, but only an agent's *model* of another agent's mental state.

Because physical distribution of agents of a multi-agent system is sometimes desirable or required, a middleware mechanism was added to the GOAL interpreter. While middleware technologies exist, these often impose restrictions on the agent model or on the execution model. To provide a minimal yet sufficient middleware for GOAL, a lightweight implementation was made using Java RMI. This implementation allows a user to launch different MAS files on different GOAL platforms, while having to configure each platform with only the hostname of the central platform.

In certain situations, a slight change of how messages are dealt with in the GOAL code is sometimes needed to deal with the relaunching of agents. These changes relate to the mailbox paradigm, and are intuitive when one thinks about the mailbox contents. Mental models are automatically removed and created as agents are stopped or launched, respectively. All in al the distribution of agents does not pose a burden on the programmer, i.e. the programmer does not have to take into account in the GOAL programs how and where the agents are distributed.

The aim of this thesis was to investigate two main questions:

1. **What is minimally required to allow communication between agents?**
2. **How easy can communication be used or applied in GOAL agents?**

To make communication between agents possible the agent programming language needs to provide programming constructs to allow the programmer to control this. Communication is seen as the sending of messages from one agent to another, so a `send` action is necessary. Agents need a way to address the messages to (other) agents. To this end, agent selectors were introduced. These messages need to be referenced by agents, so `sent` and `received` facts are kept in a message base. This *mailbox semantics* was implemented for GOAL.

The ease of use of the communication constructs was investigated by building a multi-agent system of agents that cooperate by communicating. The effort showed that programming communicating agents in GOAL is possible using the mailbox semantics. Some measures needed to be taken to control action rules that may repeat indefinitely after having received or sent a message. The `sendonce` action extended the semantics of the `send` action to facilitate these situations.

In cases where agents in a MAS may be stopped and then relaunched with the same name, and other agents believe (based on the contents of their mailboxes) that they have already interacted with these relaunched agents, some measures needed to be taken in the GOAL code to make sure that these interactions are started again. Other than that, the distribution of agents is completely transparent to the GOAL programmer.

We want GOAL agents to be able to communicate in terms of the contents of their mental states. In designing and specifying communication for GOAL, the following criteria were observed:

The communication constructs should have a well-founded theory

An important property of agent programs is their verifiability. To verify an agent program, the programming language needs to have a formal semantics. Adding communication to the language adds to the complexity of the formal semantics. Some agent technologies or standards attempt to provide a formal semantics for an ACL, but refer to the mental state of other agents in this semantics. Because of an agent's autonomy this cannot be verified and therefore the semantics is flawed. The approach taken in this thesis agrees with Singh's view that agents cannot inspect each other's mental state. Our semantics defines what a receiver of a message can *infer* about the mental state of the sender based on conversational implicatures. These inferences are aggregated in a mental model. The semantics makes no reference to the mental state of the other agent.

The distinction between beliefs and goals should persist in the communication

Since GOAL is a goal-oriented agent programming language, in which agents are programmed in terms of their beliefs and goals, it is desirable that they communicate in those terms. In other words, when one agent communicates its goal to do something or a belief in something, the receiving agent should recognize that communication as the sending agent's goal or belief, respectively. The mood operators implemented in the GOAL language allow an agent to express statements about their beliefs or goals by sending a message in a specific mood. The receiving agent can infer the sending agent's beliefs and goals from the mood and the propositional content of the message. The mood operators thus maintain the distinction between beliefs and goals in the communication.

Programming agents using the communication constructs should be pragmatic, and pose a small burden on the programmer

Using the communication constructs, the agents of a multi-agent system were implemented. The `send` action as communication primitive provides the main interface. The agent selectors give an intuitive and flexible way to address the messages to a set of agents. When using mailbox semantics, the messages could be queried from the mailbox. The mailbox semantics is intuitive, but requires some management from the programmer's part. The `sendonce` action takes care of some of that management which reduces the burden on the programmer and increases readability. Making use of the mental model queries further improves readability and intuitiveness of reasoning about the mental states of other agents.

Aside from specific situations, the distribution of agents in a MAS across multiple GOAL platforms on the same or different hosts does not affect the agent programs.

The effort of this thesis resulted in a pragmatic, usable implementation of communication in the GOAL agent programming language. Though a minimal implementation was aspired, it provides sufficient tools to program a multi-agent system of communicating and cooperating agents in a clear and pragmatic way, while providing a formal semantics of the communication constructs.

8.2 Future Work

Though the focus of this thesis was on enabling GOAL agents to communicate, some additional functionalities are worth implementing. One of these is an increased integration of the implemented middleware into the GOAL IDE. Tools that monitor and inspect agents that run on other GOAL platforms can be added. Remote launching and stopping agents together with these tools could facilitate managing a distributed multi-agent system.

Another is an improved integration of the middleware with the Environment Interface so that percepts and actions can be sent from an environment to agents on another host and vice versa (see Section 7.4). This would be interesting for use in simulated games, where two teams compete and each team is started on its own GOAL platform.

The work in this thesis mostly focused on providing the programming constructs to the programmer. On one side, of course, care was taken to ensure that the communication operations do not reduce the performance of the interpreted agent programs. On the other side, the performance benefits of having agents cooperate while running in parallel (on different machines) on multiple platform as opposed to running sequentially on one platform have not yet been measured. A testing framework could give insights in how distribution of agents effects the overall performance of the multi-agent system.

Bibliography

[1] Fabio Bellifemine, Giovanni Caire, and Dominic Greenwood, editors. *Developing Multi-Agent Systems with JADE*. Number 15 in Agent Technology. John Wiley & Sons, Ltd., 2007.

[2] Lars Braubach, Er Pokahr, and Winfried Lamersdorf. Jadex: A BDI agent system combining middleware and reasoning. In *Ch. of Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhaeuser, 2005.

[3] F. M. T. Brazier, M. Warnier, M. A. Oey, and R. J. Timmer. Agentscape tutorial, November 2008. Tutorial Given at the University of Bath and D-CIS labs, Delft.

[4] M. Dastani, J. van der Ham, and F. Dignum. Communication for goal directed agents, 2003.

[5] Mehdi Dastani. 2APL: a practical agent programming language. *Journal Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[6] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, MD, USA, 1994. ACM Press.

[7] Foundation for Intelligent Physical Agents. `www.fipa.org`.

[8] Matthew L. Ginsberg. Knowledge interchange format: the kif of death. *AI Magazine*, 12:57–63, 1991.

[9] Richard Grandy and Richard Warner. Paul grice. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2008.

[10] Robert M. Harnish. Mood, meaning and speech acts. In *Foundations of Speech Act Theory: Philosophical and Linguistic Perspectives*, pages 407–459. Routledge, 1994.

[11] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Semantics of communicating agents based on deduction and abduction. In Frank Dignum and Mark Greaves, editors, *Issues in Agent Communication*, pages 63–79. Springer-Verlag: Heidelberg, Germany, 2000.

[12] Yannis Labrou and Tim Finin. A semantics approach for kqml a general purpose communication language for software agents. In *CIKM '94: Proceedings of the third international conference on Information and knowledge management*, pages 447–455, New York, NY, USA, 1994. ACM.

[13] Michael Pendlebury. Against the Power of Force: Reflections on the Meaning of Mood. *Mind*, 95(379):361–372, 1986.

[14] Jeffrey S. Rosenschein and Gilad Zlotkin. *Rules of encounter: designing conventions for automated negotiation among computers*. MIT Press, Cambridge, MA, USA, 1994.

[15] J. R. Searle. Speech acts. 1969.

[16] John R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, January 1970.

[17] Munindar P. Singh. Agent Communication Languages: Rethinking the Principles. *Computer*, 31(12):40–47, 1998.

[18] R. Vieira, A. F. Moreira, M. Wooldridge, and R. H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. 2007.

[19] Michael Wooldridge. Semantic issues in the verification of agent communication languages. *Autonomous Agents and Multi-Agent Systems*, 3, 1999.

Appendix A

Code listings

GOAL

```
% This agent represents the coffee machine. It’s function is to supply a user
% with nice steaming fresh cups of coffee. It knows how to make coffee and
% espresso. It will communicate to find out who can make what. Notice that the
% program and perceptrules sections contain no constants, only variables.

% In fact, the program,perceptrules and actionspec implement a machine capable
% of making certain products, if it has all required ingredients, and finding
% producers of ingredients it cannot make itself.

main: coffeeMaker {
    knowledge {
        requiredFor(coffee, water).
        requiredFor(coffee, grounds).
        requiredFor(espresso, coffee).
        requiredFor(grounds, beans).

        canMakeIt(M, P) :- canMake(M, Prods), member(P, Prods).
    }
    beliefs {
        have(water). have(beans).
        canMake(maker, [coffee, espresso]).
    }
    goals {
        have(latte).
    }
    program {
        % if we need to make something, then make it (the action’s precondition
        % checks if we have what it takes, literally)
        if goal(have(P)) then make(P).
    }
    actionspec {
        make(Prod) {
            pre { forall(requiredFor(Prod, Req), have(Req)) }
            post { have(Prod) }
        }
    }
    perceptrules {
        % capability exploration:

        % ask each agent what they can make
        if bel(agent(A), not(me(A)), not(canMake(A, _))) then sendonce(A, ?canMake(A, _)).
        % answer any question about what this agent can make
        if bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prod))
            then sendonce(A, :canMake(Me, Prod)).
        % process answers from other agents
        if bel(received(Sender, canMake(Sender, Products))) then insert(canMake(Sender, Products))
            + delete(received(Sender, canMake(Sender, Products))).

        % update beliefs with those of others (believe what they believe)
        if bel(agent(A), received(A, have(X))), not(bel(have(X))) then insert(have(X)).

        % If we need some ingredient, see if we can make it ourselves
        if goal(have(P)), bel(requiredFor(P, R), not(have(R))),
           bel(canMakeIt(Me, R), me(Me)) then adopt(have(R)).
        % else try to find a maker for it
        if goal(have(P)), bel(requiredFor(P, R), not(have(R))),
             bel(canMakeIt(Maker, R), not(me(Maker))) then sendonce(Maker, !have(R)).

        % if some machine seems to need a product, tell it we have it
        if bel(agent(Machine), received(Machine, imp(have(X))), have(X))
            then sendonce(Machine, :have(X)).
    }
}
```

Figure A.1: Coffee maker agent

GOAL

```
% The Coffee Grinder is an agent capable of grinding coffee beans into grounds.
% For making grounds it needs coffee beans. Whenever it needs beans it will
% announce as much by sending an imperative "!have(beans)" to allother agents.

main: coffeegrinder {

    knowledge {
        requiredFor(grounds, beans).
        canMakeIt(M, P) :- canMake(M, Prods), member(P, Prods).
    }
    beliefs {
        canMake(grinder, [grounds]).
    }
    goals {}
    program {
        % if we need to make something, then make it (the action's precondition
        % checks if we have what it takes, literally)
        if goal(have(P)) then make(P).
    }
    actionspec {
        make(Prod) {
            pre { forall(requiredFor(Prod, Req), have(Req)) }
            post { have(Prod) }
        }
    }
    perceptrules {
        % capability exploration:

        % ask each agent what they can make
        if bel(agent(A), not(me(A)), not(canMake(A, _)))
          then sendonce(A, ?canMake(A, _)).
        % answer any question about what this agent can make
        if bel(me(Me), received(A, int(canMake(Me, _))), canMake(Me, Prod))
          then sendonce(A, :canMake(Me, Prod)).
        % process answers from other agents
        if bel(received(Sender, canMake(Sender, Products)))
          then insert(canMake(Sender, Products))
          + delete(received(Sender, canMake(Sender, Products))).

        % update beliefs with those of others (believe what they believe)
        if bel(agent(A), received(A, have(X))), not(bel(have(X))) then insert(have(X)).

        % if we want to make grounds, but have no beans, announce that we need beans
        if goal(have(P)), bel(requiredFor(P, R), not(have(R)), me(Me), not(canMakeIt(Me, R)))
            then sendonce(allother, !have(R)).

        % if some agent needs grounds, then adopt the goal to make it
        if bel(received(_, imp(have(grounds)))) then adopt(have(grounds))
         + delete(received(_, imp(have(grounds)))).

        % if some machine seems to need a product, tell it we have it
        if bel(agent(Machine), received(Machine, imp(have(X))),
         have(X)) then sendonce(Machine, :have(X)).
    }
}
```

Figure A.2: Coffee grinder agent

GOAL

```
% The milkcow is the milk cow of the coffee making process. Its primary function
% is to produce milk. It doesn’t participate in capability deliberations,
% except that it answers questions about what it can make. The cow will make
% milk whenever someone needs it, and notify that one when it is made.

main: milkcow {
  beliefs {}

  goals {}

  program {
    % be a helpful cow, see to other’s needs in milk
    if bel(received(_, imp(have(milk))), not(have(milk)))
      then make(milk).
  }

  actionspec {
    make(Prod) {
      pre { Prod=milk }
      post { have(Prod) }
    }
  }

  perceptrules {
    % (No capability exploration. Cows are generally not that interested
    % in what others can make)

    % if some machine seems to need a product, tell it we have it
    if bel(agent(Machine), received(Machine, imp(have(X))), have(X))
      then sendonce(Machine, :have(X)).

    % answer any question about what this agent can make
    if bel(me(Me), received(A, int(canMake(Me, _))))
      then sendonce(A, :canMake(Me, [milk])).
  }
}
```

Figure A.3: Milk cow agent