UNIVERSITY OF LONDON
KING'S COLLEGE LONDON

Extending agent languages for multiagent domains

by

Felipe Meneguzzi

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
School of Physical Sciences and Engineering
Department of Computer Science

June 2009

UNIVERSITY OF LONDON
KING'S COLLEGE LONDON

ABSTRACT

School of Physical Sciences and Engineering
DEPARTMENT OF COMPUTER SCIENCE

Doctor of Philosophy

by Felipe Meneguzzi

One of the most widely studied agent models is based on the notions of beliefs, desires and intentions (or BDI) as mental attitudes that guide the selection of courses of actions. However, BDI agent languages have been used mostly in the context of single agents based on a plan library of behaviours invoked reactively and, though they provide a theoretically sound basis for agent development, they offer limited support for multiagent systems with dynamic plan libraries.

In particular, when new plans not foreseeable at initial design time are required, the agent must be redesigned. Moreover, when designing multiagent systems, agent languages provide at most a communication language with no other consideration of interaction.

This thesis aims to address these limitations by introducing a new agent language and architecture that includes a mechanism for processing goals in a manner that decouples goal achievement from plan execution, as well as generating new plans to cope with unforeseen situations at design time. It bridges the gap between agent languages and multiagent systems by introducing a simple cooperation mechanism together with a norm processing mechanism aimed to providing some degree of societal control.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

Many people contributed in one way or another to my success in the course of researching for and writing this thesis, thus thanks are in order to the following people.

First and foremost, to my supervisor Michael Luck, whose attention to detail has made me see so many things to be improved that I might not have seen by myself. If this thesis is in a good state, it is certainly because of him, any errors or omissions are entirely my fault.

To my second supervisor Andrew Jones.

To the anonymous reviewers of AAMAS, DALT, CEEMAS and AT2AI whose feedback on our papers allowed us to refine the research in this thesis.

To Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for financing my doctoral studies in England, and particularly Vanda Lucena, the person in charge of my scholarship.

To my colleagues at King's College London, Sanjay Modgil, Simon Miles and Nir Oren, with whom I had the opportunity to discuss my research and who provided me excellent input throughout my time at King's.

To my former colleagues and lab mates at the University of Southampton, Maíra Rodrigues, Luke Teacy and Jigar Patel for their friendship in the initial uncertain moments of my PhD.

To Rafael Bordini and Jomi Hübner, the developers of Jason, for their support in the use of their platform.

To my girlfriend Ana María Miranda: I went to England to get a PhD and found so much more in her.

To my parents Nelso Meneguzzi and Maria Rech Meneguzzi and siblings Clarissa and Cristiano for having given me a stable family environment and supported the education that allowed me to give my first steps into science, which led me to this PhD.

To Ana María

Physics is like sex. Sure, it may give some practical results, but that's not why we do it.
– Richard Feynman

Chapter 1

Introduction

1.1 Computing and Interaction

As computer science has evolved during the last century, our understanding of computation has expanded from a mathematics-centred model focused on data processing to one based on interaction among distributed entities [Wegner and Goldin, 2003]. This evolution of the notion of computation has come about not only through advances in hardware but also through advances in abstraction mechanisms needed in the development of ever more complex programs. As researchers and developers departed from the scripted batch calculations of early computers, the need for higher-levels of abstraction became more and more important to insulate developers from the complexities of lower level hardware control. This process of evolution was initiated with the very first programming languages, which delegated decisions about lower level machine instructions to a compiler, and have now reached a level of sophistication that is difficult for a human to beat in terms of efficiency.

This tendency of delegating more and more activities to computer systems continues today. Computers no longer exist to be used exclusively for batch processing, but instead exist in a large networked environment as entities that interact not only with other computers but also with human users. As a consequence, the traditional understanding of computation as data processing-centred has to be revised to cope with its more interaction-oriented contemporary usage.

While low level machine instructions are the objects of abstraction in traditional computing, interaction-based computing requires that other elements be abstracted. Such a need was identified very early in the history of artificial intelligence by Alan Turing, whose *Turing Test* is an eminently interactive computing exercise. Turing stated:

> "Our problem then is to find out how to programme these machines to play the game. At my present rate of working I produce about a thousand digits

of programme a day, so that about sixty workers, working steadily through the fifty years might accomplish the job, if nothing went into the wastepaper basket. Some more expeditious method seems desirable."

[Turing, 1950]

We agree that it is desirable for appropriate methods and tools to be developed if computing based on interacting entities is to succeed. Indeed, many researchers today argue that the notion of autonomous agents is the most intuitive way of describing interactive systems [Jennings et al., 2006; Wegner and Goldin, 2003]. From a software engineering perspective, agents can be seen as the evolutionary successor to objects [Odell, 2002]. In a rough analogy one could say that while objects have attributes and methods, agents have mental states and plans to achieve their objectives. From a practical perspective, agents are a better abstraction because they encapsulate not only data (mental state), but also the process of selecting which behaviours are needed and when.

Before we start a discussion of how agents can be used to help solve the problem of designing interactive systems, we must review the notion of agent. The most widely accepted, though vacuous, concept of agents states that an agent is an entity that perceives a certain environment through sensors and acts upon it through actuators. Within the artificial intelligence community, a more elaborate view is that of Wooldridge and Jennings [Wooldridge, 2002; Wooldridge and Jennings, 1995], which state that an agent is a computer system *situated* in an *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives. This view is further refined, attributing to agents the following characteristics:

- autonomy;
- social ability;
- reactivity; and
- pro-activeness.

Autonomy means that agents have control over their internal state and behaviour and thus operate without intervention from external entities, be they humans or other agents. Social ability means that agents interact with other agents (including humans) using some kind of agent communication language. Reactivity means that agents respond in a timely fashion to events in their environment. Finally, pro-activeness means that agents take the initiative to act instead of just reacting to events in the environment. Thus, an autonomous software agent is expected to take the initiative (autonomous action) in doing some useful computation on behalf of a human user, interacting with other software and human agents to further whatever goal it is assigned to achieve (its design objectives).

When considering concrete agent implementations, these four characteristics have interesting implications if taken together. Autonomy and reactiveness imply that an agent not only has to choose its own behaviours, but must do so in a timely fashion. That is, an agent must not take so long to make a decision that the decision is no longer relevant. In particularly busy environments this poses a challenge since, with no further information, an agent must react in a timely fashion to relevant events. Furthermore, if an agent must not only be reactive, but also proactive in trying to achieve long term goals, instead of just addressing immediate needs, it needs to prioritise certain activities. Social ability, in Wooldridge's view [Wooldridge, 2002], seems to be limited to the use of some agent communication language, implying the need for just the means of communication, rather than the ability to autonomously and pro-actively seek out help when necessary and supplying help when convenient. We believe that the tools used in the creation of interacting agents need more than that to be useful abstractions for agent design.

The study of agents includes many aspects both theoretical (*e.g.* philosophy of beliefs and truth, societal dynamics) and practical (*e.g.* architectures, languages). Two of the most important aspects of the study of practical agents are: understanding the organisation and internal processes required for agents to operate in real computer systems (*i.e.* agent architectures); and the abstractions through which a designer describes individual agents while avoiding the need to deal with complexity from the underlying agent processes (*i.e.* agent languages). In this thesis, we will chiefly be concerned with the disconnect between these practical issues and the theoretical properties of agents.

1.2 Multiagent Systems

Agent-based software has been advocated as an ideal technique for the development of large, distributed applications, viewing them as a number of independently controlled parts that interact and cooperate to achieve their design objectives. Much research dealing with *agent languages* has focused on the description of plans used by an *individual* agent to interact with the world [Bordini et al., 2007; Brooks, 1986; Dastani et al., 2005; Howden et al., 2001]. Although in multiagent systems, agents are assumed to be able to use interaction to achieve goals, agent languages seldom provide mechanisms to do so, and cooperation is generally developed in an *ad hoc* fashion. Even when cooperation *is* involved, it tends to use a highly specialised version of any of a number of existing cooperation techniques, assuming a distributed but ultimately *predefined* set of abilities in the society.

Cooperation is often cited as one of the main characteristic properties of multiagent systems [d'Inverno et al., 1997; Doran et al., 1997], yet there are several different modes of cooperation that can be identified:

- multiple agents acting towards a common joint goal;

- one agent acting to achieve goals for another agent; and
- agents synchronising their actions so as to avoid negative interference.

The first, and most common mode of cooperation in agents consists of a group of agents sharing a possibly implicit joint goal and acting to achieve this goal in a coordinated way. This goal might be negotiated at runtime or exist in all agents by design. The second possible mode of cooperation consists of one or more agents performing actions that are not directly related to their own goals, but rather support the achievement of the goals of another agent. The third and final mode of cooperation commonly considered consists of agents agreeing on some coordination of their individual actions towards their individual goals in such a way that no agent jeopardises the operation of another. In order to address cooperation, however, we need to consider individual agents themselves, thus taking into consideration existing single agent languages.

1.3 Declarative Goals

Research on agent architectures has yielded a number of models for individual agent operation, among which one of the most popular is inspired by a philosophical model of reasoning based on the three mental components of beliefs, desires and intentions (BDI) [Bratman, 1987]. This model postulates that autonomous agents have a model of the world represented as beliefs, as well as a set of desires constituting potential objectives, and intentions representing commitment to particular courses of action to achieve particular desires.

BDI architectures and models tended to avoid including many of the declarative aspects of desires/goals in support of practicality. More specifically, the first instances of complete BDI logics [Rao and Georgeff, 1995b] assumed an agent able to foresee all of the future ramifications of its actions as part of the process of deciding which courses of action to take [Schut and Wooldridge, 2001]. This assumption was clearly too strong if computationally-bounded BDI architectures were to be constructed. Therefore, when designing practical architectures based on specific BDI logics, modifications were necessary to avoid unbounded computations.

As a consequence, such architectures have tended to use plans with implicit goals for practical reasons. In these architectures goals are implicit in the sense that instead of actually evaluating the world and predicting which of its behaviours will bring about a desired outcome, an agent associates certain events in the environment with the enaction of particular behaviours. This approach results in very good performance as far as computational efficiency is concerned, but moves away from the ability to adapt responses to unforeseen events in the world. The idea here is that, since an agent cannot look directly into future world-states and then select the sequence of actions that leads to the desired future (as this would imply omniscience), the inverse approach is taken; that is, an agent selects, from a set of known courses of action, the one that would lead to the desired future. In practice, this means that an agent no longer selects directly what it desires to achieve, but rather what it desires to perform under the assumption that its actions ultimately bring about the desired state of affairs. These goals are commonly known as goals *to do* or procedural goals [Winikoff et al., 2002].

Alternatively, it is possible to design agents that select a desired state of affairs directly, using some process to find plans that achieve this desired state of affairs. These explicitly desired states of affairs are commonly known as goals *to be* [Winikoff et al., 2002] or declarative goals. As a consequence, the actions required by the agent to reach such a state of affairs are decoupled from the ultimate goal. This gives rise to the problem of discovering which actions will have to be taken by the agent to realise its goals. The most widely known BDI agent implementations bypass this problem through the use of plan libraries where the courses of action for every possible objective are stored [d'Inverno et al., 2004; Ingrand et al., 1992; Rao, 1996], and which we have seen are associated with goals to do. The near absence of pragmatic architectures that implement the notion of goals to be represents a gap that current research is trying to address.

1.4 Research Objectives

It should be clear that agents are a powerful abstraction for describing interactive systems, and that the BDI model is a very natural abstraction of reasoning, and therefore interaction, involving human agents. It is also the case that, in order to build agent systems, adequate tools must be provided that allow the theoretical properties of agent systems to be implemented in concrete computer systems. However, there seems to be a pronounced disconnect between agent theory and the existing architectures and languages, and the systems resulting from these tools do not fully reflect the potential afforded by the agent paradigm.

In response to the above issues, this thesis aims to provide a general purpose agent language by bridging the gap between the possibilities postulated by agent theory and their availability in existing agent languages. It is important for flexible agent languages to provide mechanisms that allow agents to select their desired state of affairs, allowing an agent to achieve goals independently from particular courses of action. Moreover, if an agent is to achieve its goals flexibly and autonomously, it must be able to generate new plans beyond those defined initially by the designer. Thus, our aim is to include mechanisms that facilitate the development of each of the four characteristics of agency: autonomy, reactivity, pro-activeness and social ability. The mechanisms we propose to develop are summarised in Table 1.1.

Characteristic	Mechanism
Autonomy	Planning
Reactivity	Meta-reasoning
Pro-activeness	Declarative Goals
Social Ability	Cooperation and Norms

TABLE 1.1: Summary of agent characteristics and associated proposed mechanisms.

Declarative goals allow a designer to create agents that decouple the execution of its plans from goal achievement, thus enabling the resulting agents to try different strategies at runtime without the need to consider every possible event happening in the environment. We argue that declarative goals are the best way of representing long-term objectives, without which no pro-activeness can exist. Therefore, the representation of declarative goals is a vital component of any general purpose agent language. Processing of declarative goals, however, requires some changes in the way in which traditional BDI-style agent languages operate. In particular, an agent needs to choose a plan of actions to achieve the desired world-state, which is traditionally done using a library of predefined plans. However, for an agent to be truly able to operate autonomously, it must be able to handle new situations without assistance, so the ability to generate new plans at runtime is crucial for an agent language interpreter.

In order to act effectively in any complex environment, autonomous agents must have control over their internal state and behaviour. To exercise this control an agent needs some means of reasoning about its internal state, often in a process known as meta-level reasoning (or meta-reasoning). This is higher level reasoning about the reasoning process itself, and in agent systems it is commonly used in enforcing rationality in the choice of goals and actions performed by an agent, ensuring that it behaves as effectively and efficiently as possible. Through meta-reasoning an agent is able to explicitly consider goals before committing to them, and consider courses of action before executing plans, in opposition to simply reacting to events in the environment.

A truly autonomous agent has a broad spectrum of options when deciding what goals to achieve and how to achieve them. However, traditional agent languages tend to simplify the process of goal selection and subsequent plan selection to a trigger-response mechanism. This trigger response mechanism is tightly coupled with a plan-library based approach to agent design. Any prioritisation is implicit in the plan library through carefully crafted event triggers and possibly some additional contextual information. This results in an agent having behaviour patterns set at design time, limiting runtime flexibility. If too many of the triggers in a plan library are activated at the same time, an agent can easily become overwhelmed and lose reactivity. In procedural languages, specifying meta-reasoning separately from the plans removes the need to replicate internal management code throughout the plan library, facilitating development. By contrast, declarative architectures are defined by desired states to be achieved, and capabilities with which an agent can achieve them, where an interpreter selects capabilities to achieve goals, and goal conflict resolution must be performed by this interpreter. In declarative languages, the lack of some goal selection policy means that goals and plans are selected arbitrarily, since in theory the designer does not specify precisely how goals are to be achieved. We argue that models of motivated behaviour [Mele, 2003] can provide a valuable abstraction for the specification of meta-reasoning, specifically in the context of the BDI model. Furthermore, if an agent needs to quickly prioritise goals and actions, motivation-based rules provide an efficient and rational way of doing this. We thus aim to address this limitation in the selection of goals by adding a meta-reasoning component to an agent's description, leveraging it as both a mechanism of goal generation and of prioritisation.

Finally, although social ability is a key characteristic in multiagent systems, agent languages are notoriously poor in providing mechanisms to facilitate the development of cooperative systems. We aim to address this shortcoming by providing a language-level mechanism that supports an agent in carrying out plans on behalf of another. Now, if an agent is able to request for another agent to execute plans on its behalf, the abilities of the requesting agent are effectively expanded to include those of the both agents. These new capabilities can thus be used to compose new multiagent plans. Moreover, systems of autonomous agents need some degree of behavioural predictability, and to this effect, a general purpose language for multiagent systems requires some basic norm processing ability. Although normative systems have received a lot of attention at the *macro* level, that is at the level of society, the *concrete* effects of deciding to follow norms *within* an agent's reasoning cycle have received comparatively little attention. Thus, it is important to investigate how norms must affect the reasoning of complying agents.

In summary, therefore, our aims are as follows.

- To provide agent architectures with practical plan generation capabilities, allowing them to adapt to new circumstances not foreseen at design time.
- To provide a meta-reasoning component for agent architectures so they are capable of autonomously prioritising goals and adjusting priorities at runtime.
- To introduce language-level support for agent cooperation that takes into consideration plan generation capabilities allowing non-scripted cooperation.
- To develop a means to consider societal constraints in order to bound an agent's possible behaviours resulting from non-scripted cooperation.

1.5 Research Contributions

In carrying out the research necessary to accomplish the objectives of Section 1.4, we have made a number of contributions to the field of agent systems. Thus, the contributions of this thesis can be enumerated under different areas.

Action-directed reasoning: In terms of our aim of adding plan generation capabilities, we have taken a traditional agent language, namely AgentSpeak(L), and extended it to allow agents to generate new plans by combining existing plans within their plan libraries. The resulting contributions are the following.

1. Development of a declarative-goals enabled BDI agent, based on AgentSpeak(L). This allows agents to be defined in terms of desired goals that are decoupled from the means through which they are achieved, facilitating the design of pro-active and autonomous agents.
2. Development of AgentSpeak(PL), consisting of AgentSpeak(L) enriched with planning capabilities geared towards the achievement of these declarative goals, allowing agents to compose new plans at runtime.
3. Construction of a mechanism to allow an agent to *reuse* newly created plans through the generation of a *context condition*, thus amortising the computational effort expended in planning. This provides a further refinement of the AgentSpeak(PL) system.

The contributions on plan generation and declarative goals have been published as:
[Meneguzzi and Luck, 2007a] Felipe Meneguzzi and Michael Luck. Composing high-level plans for declarative agent programming. *Proceedings of the Fifth Workshop on Declarative Agent Languages*, pages 115–130, 2007.

The work on plan reuse has been published as:
[Meneguzzi and Luck, 2008b] Felipe Meneguzzi and Michael Luck. Leveraging new plans in AgentSpeak(PL). In Matteo Baldoni, Tran Cao Son, M. Birna van Riemsdijk, and Michael Winikoff, editors, *Proceedings of the Sixth Workshop on Declarative Agent Languages*, pages 63–78, 2008.

Meta-reasoning: Creating new plans at runtime introduces the problem of managing plans at the meta-level; that is, multiple plans may interact in the agent's reasoning cycle, and cause difficulty from their interactions. In particular, when a plan library is statically specified by a designer, interaction among plans can be predicted and accounted for in the plans themselves, but when new plans can be created at runtime, the decision to execute a certain plan needs to take into consideration its effects on other plans. Thus, our work in this area consists of the specification of meta-reasoning through a *motivation* abstraction, including the following contributions.

1. Construction of a new model for motivations in a declarative goal-based architecture. This work builds on existing work to tie in to a wealth of prior research, making the resulting mdBDI model widely applicable.

2. Development of a new, generic, language for specifying meta-reasoning strategies as motivational functions, that enables motivation-based meta-reasoning to be addressed in a domain-independent fashion.

3. Development of a mechanism that uses the specified motivations to prioritise goals at runtime, ensuring reactivity. This mechanism is used to provide a further architecture, AgentSpeak(MPL), as a refinement of the previous architecture.

The work on motivation-based meta-reasoning has been published as:
[Meneguzzi and Luck, 2007b] Felipe Meneguzzi and Michael Luck. Motivations as an abstraction of meta-level reasoning. In Hans-Dieter Burkhard, Gabriela Lindemann, Rineke Verbrugge, and László Z. Varga, editors, *Proceedings of the 5th International Central and Eastern European Conference on Multi-Agent Systems*, volume 4696 of *LNAI*, pages 204–214. Springer, 2007.

Cooperation: Concerning our goal of addressing the lack of agent language-level cooperation mechanisms, we develop a cooperation mechanism based on the willingness of cooperating agents to execute plans on behalf of other agents. This cooperation mechanism leverages the planning mechanism from the previous contributions to create multiagent plans, by isolating the distributed aspects of the plans from cooperating agents from the planner. As a result, this ensures that the same basic architectural components can be used, increasing applicability. It includes the following contributions.

1. Development of an AgentSpeak(L)-based mechanism to *discover* the capabilities of cooperating agents.

2. Creation of a method of using the discovered capabilities in cooperative plans through the use of local *proxy plans* that abstract communication and coordination with other agents.

3. Generation in AgentSpeak(PL) of *cooperative plans* based on these proxy plans.

4. Provision of a *failure handling* mechanism to eliminate unreliable cooperative plans.

The work on this cooperation method has been published as:
[Meneguzzi and Luck, 2008a] Felipe Meneguzzi and Michael Luck. Interaction among agents that plan. In Bernhard Jung, Fabien Michel, Alessandro Ricci, and Paolo Petta, editors, *Proceedings of the Sixth International Workshop: From Agent Theory to Agent Implementation*, pages 133–140, 2008.

Normative reasoning: Now, creating new plans at runtime, both by single and by cooperating agents, creates the possibility of undesired behaviours emerging in a society, necessitating a mechanism of societal control to bind the agents to acceptable behaviours. We address this issue by extending our agent language to cope with societal constraints through the provision of norm processing capabilities. The ensuing contributions are the following.

1. Development of a mechanism to enable an agent to dynamically *suppress* plans that lead to prohibited behaviours when such prohibitions come into force.
2. Similarly, provision of a mechanism to *restore* previously suppressed plans once prohibitions cease to be in force.
3. Development of a mechanism, leveraging the planning capability to create new plans, to achieve the stipulations of obligations, thus complying with them.

The work on normative reasoning has been accepted for publication as:
[Meneguzzi and Luck, 2009] Felipe Meneguzzi and Michael Luck. Norm-based behaviour modification in BDI agents. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, page (to appear), 2009.

1.6 Methodology

The research carried out to achieve the various contributions enumerated in Section 1.5 relies on different methodological approaches, reflecting the type of contribution being sought. The contributions relating to plan generation consist of not only new language constructs, but also a modification to the agent's reasoning cycle; as a consequence, the validation of this contribution includes examples of how the resulting language can be used as well as empirical tests to assess the impact on computational cost of the planning process within the agent reasoning cycle. Likewise, our contributions towards the inclusion of meta-reasoning in AgentSpeak(L) result in the introduction of language constructs and modifications to the agent's reasoning cycle, and thus their validation consists of examples of language use, and an empirical evaluation of the efficiency gains that can be attained through the use of such meta-reasoning.

Unlike the two previous contributions, however, the introduction of a cooperation mechanism in AgentSpeak(L) is difficult to evaluate quantitatively, or to compare quantitatively with the original architecture. In consequence, our validation effort consists of example applications of the techniques developed, demonstrating what can be achieved using them. Similarly, the norm processing mechanism is also difficult to evaluate in a quantitative empirical analysis, and we therefore demonstrate it through examples of the effects of the mechanism on concrete AgentSpeak(L) agents.

1.7 Thesis Overview

This thesis is structured as follows: Chapter 2 reviews literature relevant to the work developed in this thesis with an emphasis on agent systems, motivated reasoning, meta-level control and norms, ending with a discussion of how these topics establish a set of requirements for this thesis; Chapter 3 expands on Section 1.3 introducing the notion of declarative goals in traditional agent languages and the use of planning to allow flexible goal achievement; Chapter 4 expands on the notion of meta-level reasoning discussed in Section 1.3 and describes the motivations-based meta-reasoning mechanism we introduce into our agent language; Chapter 5 expands on Section 1.2 detailing the cooperation strategy supported by our agent language; Chapter 6 elaborates the notion of social ability provided in Chapter 5 and describes the norm processing mechanism introduced in our language; and Chapter 7 concludes this thesis by summarising our contributions pointing out limitations and directions for further research.

Chapter 2

Agent languages and architectures

As stated in Chapter 1 existing agent languages and architectures are very limited and do not typically address issues of plan generation, meta-reasoning and multiple self-interested agents in non-scripted cooperation in an adequate fashion. Our aim in this thesis, therefore, is to seek to find ways to augment agent languages and their underlying architectures with such qualities by providing a means to generate new plans dynamically (for flexibility and reuse), to cooperate with other agents (for multi agent systems), to constrain agent actions for societies (to ensure stability in multi agent systems), and to do all this in a self-motivated and autonomous manner.

In this chapter, we are therefore concerned with reviewing agent languages and architectures and elements that have been considered in isolation to be important for autonomous agents. We start by reviewing notions of agents in Section 2.1, followed by a number of notable agent architectures in Section 2.2 and languages in Section 2.3. Clearly, there is a very large body of work on agent architectures and languages, with far too many different types to review exhaustively here. In what follows, therefore, we concentrate on reviewing a representative selection of architectures, to cover the broad range and give an indication of the kind of work that has been previously undertaken to identify achievements and limitations. When relevant to the specifics of our work, however, we provide more substantial coverage as appropriate. Having laid down the basics of agent systems and languages, we proceed to describe models of meta-reasoning in Section 2.4 and multiagent systems in Section 2.5. After considering the state of the art in terms plan generation, meta-reasoning and multiagent systems, we conclude the chapter with a discussion of the shortcomings encountered in individual languages, architectures and techniques, pointing to requirements for agents to be developed throughout the remainder of the thesis.

2.1 Agents

In recent years, use of the term *agent* has become widespread in computer science, while the notion of *autonomous agent* has also received some attention in modelling complex systems. Regardless of the popularity of the term, loose definitions of agent are abundant. However, the only widely accepted definition of an agent is that of an entity *that perceives a certain environment through sensors and acts upon it through actuators*, as illustrated in Figure 2.1. Through this definition almost any computer program or device can be considered an agent, and this has been done by many. For example, a thermostat can be seen as an agent [Franklin and Graesser, 1996], without improving the understanding of the system in any way, and therefore being used trivially [Shoham, 1993].

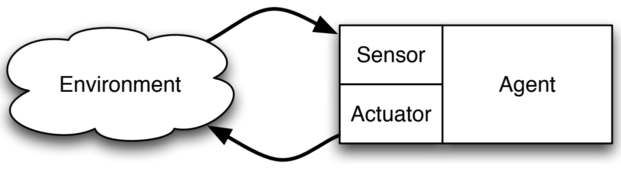


FIGURE 2.1: Less than helpful definition of an agent.

We can consider agents as a higher-level of abstraction than traditional software engineering techniques that facilitates the construction of complex systems, but this simple definition of agent provides no explanation as to how the *agent* box in Figure 2.1 processes perception and generates action. By contrast, we have seen that Wooldridge and Jennings provide a more comprehensive view [Wooldridge and Jennings, 1995], stating that an agent is “a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in this environment in order to meet its design objectives.” This later definition puts together two key characteristics that we shall explore further in this chapter: first, the emphasis on the association of an agent to an environment; and second, the idea that agents have certain objectives that they must attain using *autonomous* action. Autonomy, in a very broad sense, means freedom of action [Soanes and Hawker, 2005], and in the agent literature it generally means that an agent initiates its actions without direct prompting of a user or programmer intervention [Luck et al., 2004].

In order to make the concept of agent more helpful in allowing us to build these complex systems, agents need to have architectures based on some theoretical grounding rather than *ad hoc* programming. These architectures should contain a somewhat intuitive process that leads from sensor input to actuator action while abstracting a lot of the lower level concerns needed in the creation of traditional systems, otherwise the resulting agent abstraction is no better than traditional software engineering techniques. Therefore, a number of agent architectures have been proposed in order to allow the creation of agents that are not simply *ad hoc* implementations, but general frameworks for describing and building autonomous agents. In the following sections we survey a number of agent architectures as well as programming languages based on the notion of autonomous agency.

2.2 Agent Architectures

In this section we review some of the most important agent architectures in relation to this thesis. We start by providing a taxonomy of agent systems that allows us to pick representative architectures of each type of agent system. We then briefly review key characteristics of these architectures in the following sections.

2.2.1 Taxonomy

A number of taxonomies have been proposed in order to classify the many existing agent theories and implementations. Such typologies are based on diverse criteria, one of the most common of which is the *purpose* of an agent. For example, Nwana [Nwana, 1996] classifies agents as interface agents (to create a flexible GUI), information agents (to filter news or email for a user), among others. In contrast, others, such as Franklin and Graesser [Franklin and Graesser, 1996], use the most notable capability in an architecture as their criterion: for instance, an architecture in which learning plays a predominant role makes it a *learning* architecture, an architecture that can change its set of behaviours at runtime is a *flexible* architecture, or an architecture in which considerable emphasis is given to moving agents among computers is a *mobile* architecture, and so on.

These types of taxonomies are limited because they are connected to the particular attribute being emphasised at the time the classification was developed. Thus, such taxonomies are vulnerable to obsolescence together with the technologies that justified them. For example, mobility is no longer a *defining* aspect of an agent architecture, and though this does not invalidate them completely, it shortens the time in which they are relevant.

On the other hand, there are classifications in which the criterion is the internal agent architecture, of which perhaps the best known classification was articulated by Wooldridge [Wooldridge, 1999]. In this classification agents are organised according to their architecture, enumerating abstract and concrete architectures. As abstract architectures, Wooldridge lists:

- **purely reactive agents**, whose actions at any given time are determined simply by the sensor input at that time; and
- **agents with state**, which maintain an internal state that is modified by sensor input at each point in time and whose actions are the result of a function over this internal state.

This classification is interesting, but it does not give enough emphasis to individual types of state-storing architectures, which encompass a very large class of agents. Agent architectures with state have a number of notable ways of encoding and manipulating their internal states. Therefore, refining this into more concrete architectures, Wooldridge [Wooldridge, 1999] enumerates:

- **logic-based architectures**, in which aspects of the world are modelled in logic and where decision-making occurs through a theorem prover;
- **reactive architectures**, in which decision-making is achieved through some sort of direct mapping between situation and action;
- **belief-desire-intention architectures**, in which decision-making is the result of the manipulation of data structures representing mental states in an agent, especially beliefs, desires and intentions; and
- **layered architectures**, in which control and reasoning is distributed through various software layers that reason at different levels about the environment and the agent's knowledge.

Such a categorisation is interesting as it follows the chronological evolution of architectures. However, we disagree with a specific category being given to logic-based architectures, since most belief-desire-intention architectures use logic at some point, and even layered architectures use some form of logical representation, at least of a database. Thus, the category of logic-based architecture seems to be a part of other architectures. In consequence, we use the principles of Wooldridge's taxonomy to organise agents into three different categories, namely: reactive, deliberative and hybrid agents, as illustrated in Figure 2.2. Reactive agents include the subsumption architecture and Pengi, among others; deliberative agents include all BDI-architectures as well as BOID; and hybrid agents include TouringMachines and InteRRaP.

We outline key aspects of reactive architectures in Section 2.2.2, while deliberative architectures include the large class of belief-desire-intention inspired architectures, described in Section 2.2.3, as well as BOID agents, described in Section 2.2.4. Finally, significant hybrid architectures described in this chapter include TouringMachines and InteRRaP in Section 2.2.5.

2.2.2 Reactive Architectures

Reactive architectures were created as an attempt to overcome certain limitations perceived by some researchers [Brooks, 1986] towards traditional symbolic architectures, among which that:

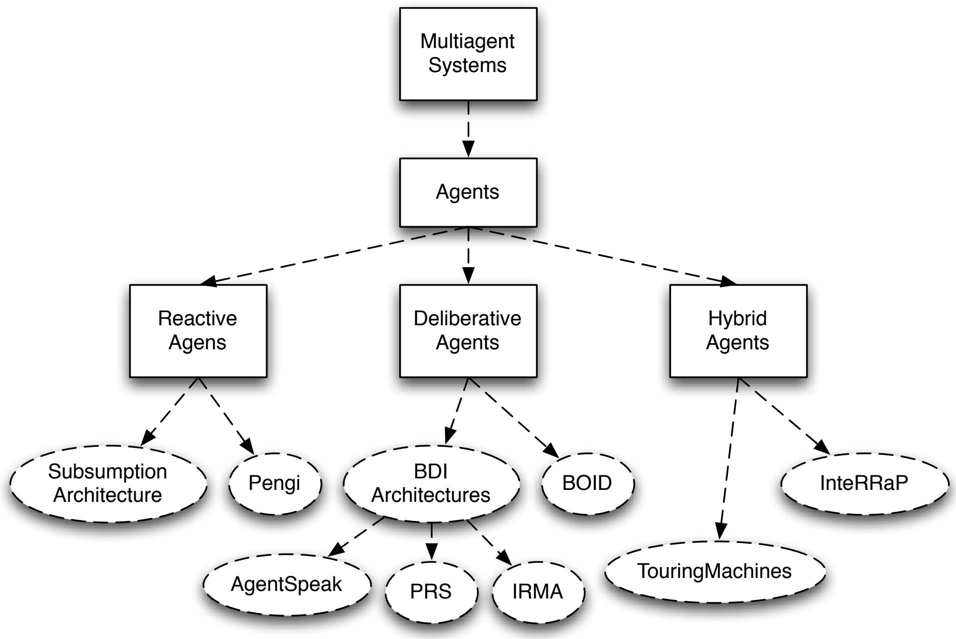


FIGURE 2.2: Our taxonomy.

- it is not necessary to create a symbolic representation of the world, nor are syntax-based inference methods necessary for decisions to be made;
- intelligence is not disembodied, but is instead a result of an agent's interaction with the world; and
- intelligent behaviour emerges from the interaction of many simpler behaviours.

The resulting reactive architectures, including the Subsumption architecture [Brooks, 1986] and Pengi [Agre and Chapman, 1987], demonstrate the possibility of achieving fairly complex behaviour without using symbolic models in the reasoning process. Nevertheless, more complex applications are not trivial to design using the reactive model, mainly due to intelligent behaviour having to emerge from a set of simpler behaviours. Reactive architectures are also lacking when proactive behaviour, in which an agent seeks to achieve longer-term objectives, is necessary, since it must avoid being constantly diverted from its aims by responding to environmental changes.

2.2.3 BDI Architectures

In contrast to reactive architectures, one of the most widely studied agent models is based on the notions of *beliefs*, *desires* and *intentions* (or BDI) as mental attitudes that guide the selection of courses of action. In this model, beliefs describe knowledge about the world, while desires are states of affairs to achieve, and intentions are commitments to achieving a particular subset of desires. The BDI model has its origins in the philosophical work of Bratman [Bratman, 1984] to explain the way in which humans select a series of actions directed at the achievement of a larger goal while avoiding spending time considering less important ones.

This philosophical model inspired the idea of intelligent agents using the same mental abstractions of beliefs, desires and intentions to describe the operation of computer systems. The first concrete BDI agent architecture was the Intelligent Resource-bounded Machine Architecture (or IRMA) developed by a team including Bratman himself [Bratman, 1987], and was created to demonstrate the viability of the BDI model of practical reasoning. IRMA subsequently evolved into formalisations [Cohen and Levesque, 1990] and a more complete computational theory [Rao and Georgeff, 1995a].

Following IRMA, Georgeff and Lansky created the Procedural Reasoning System (PRS) aiming at a BDI architecture suited for real world applications [Georgeff and Lansky, 1987]. It was first used in the implementation of a task control system for a NASA spacecraft simulator.

A PRS agent or module consists of four components:

- a *database* containing the current system's beliefs about the world;
- a set of current *goals*;
- a procedure or plan library (*knowledge area (KA) library*); and
- an *intention structure.*

KAs describe action and test sequences intended to achieve the proposed goals or to react to specific situations [Ingrand et al., 1992] while the *intention structure* maintains the set of plans chosen at runtime for execution. PRS integrates these components via an *interpreter*, which functions as an inference mechanism that manipulates them and selects an adequate plan based on the system's beliefs and goals, putting this plan into the intention structure and executing it. Here, the idea of real world applications implies that an agent needs to react quickly to a dynamic environment, and PRS emphasises computational efficiency in execution. In order to accomplish this and, just as in reactive architectures, PRS drops the ability to generate new plans, relying instead on a library of *procedural* plans. PRS and its successor, the distributed Multi Agent Reasoning System (dMARS) [d'Inverno et al., 2004], were subsequently used to create various practical applications [Georgeff and Ingrand, 1989b; Ingrand et al., 1992]. Finally, the language used to describe the agents was formalised by Rao [Rao, 1996] in AgentSpeak(L), which we shall go in further detail in Chapter 3.

In this way, BDI agent architectures have been used mostly in the context of single agents based on a fixed plan library of behaviours invoked reactively. These architectures provide a theoretically sound basis for agent development but offer limited support for developing multiagent systems with *dynamic* plan libraries; that is, an agent has no way of adapting its plan library after being deployed. In particular, when an application requires agents to create new plans to cope with circumstances not foreseeable at design time, a designer is required to develop an *ad hoc* solution to the problem.

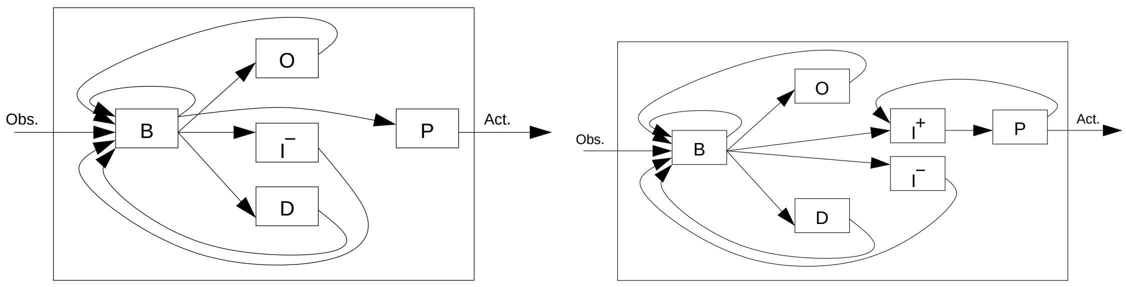
2.2.4 BOID Architecture

The Beliefs-Obligations-Intentions-Desires architecture (BOID) was created by Broersen *et al.* [Broersen et al., 2001] aiming at the construction of *normative* agents (*i.e.* agents in which some of its goals are adopted as a result of commitment to social norms). The architecture is based around a framework in which each mental attitude is represented as a component that processes generic logical formulas in a sequential loop until an agent commits to an action. An initial set of formulas is generated as a result of observation from the sensors and supplied as input to the first component, which processes the formulas in a loop until no modifications are made. The component then passes the formulas onwards until the last component does its processing, which ultimately results in commitments. Each set of formulas processed by a mental-attitude component is called an *extension* [Broersen et al., 2001].[1] These components take as input one set of logic formulas and provide as output another set of formulas that have been calculated based on a set of inference rules encoded in the component.

Broersen *et al.* make an extensive evaluation of the potential conflicts among the mental attitudes (for example, desires may conflict with obligations) concluding that, depending on the type of agent being considered, a different priority of conflict resolution is necessary. So, the idea behind the component-based calculation in BOID is that mental-attitudes (or components) that have higher priority in the conflict resolution hierarchy perform their calculations first, feeding the output to the subsequent component. A component is said to be without conflict regarding a previous component in the chain if the set of formulas it receives as input remains unmodified as the output. If the set is modified, it is fed back to the beginning of the chain. Ultimately, this processing of formulas occurs so that BOID agents consider all effects of actions before committing to them.

The sequence in which the mental components are organised for processing these formulas determines a different type of agent so, for example, an agent in which the desires component comes before the obligations component is called a *selfish* agent, because desires have priority over obligations. Similarly, realistic agents are those that consider beliefs the most important component, where the initial set of formulas is first given as input to the beliefs component and then to the other components. Agents of this kind typify *BOID* agents, which are also endowed with a *planning* component that takes a set of formulas as input and generates a set

[1]Since an extension in BOID is just an opaque nomenclature for sets of logic formulas, we use the term *set of formulas* here for clarity.



(a) BOID Architecture. (b) Multiple Extension BOID.

FIGURE 2.3: Variations of BOID [Broersen et al., 2001].

of actions scheduled to be performed. This organisation is illustrated in Figure 2.3(a), where B represents the beliefs component, D the desires component, O the obligations component, P the planning component and I^- represents the set of previous *intentions*, or the actions to which the agent has committed in the previous processing cycle. Figure 2.3(b) shows a modification of the basic BOID architecture that allows multiple sets of formulas to be used in the generation of plans. Here, all new intentions are processed by I^+ (emphasising that these are newly selected intentions) ordered by some preference relation, and then given as input to the planning component. If for any reason the selected formulas (*i.e.* intentions) cannot be translated into a feasible plan, the I^+ component sends the next best formulas to the planning component.

Although BOID contains an interesting experiment on the logics driving many types of reasoning within a BDI architecture, it is a very difficult architecture to understand, limiting its attractiveness as a tool for the development of practical agents. Moreover, the very nature of the processes used by an agent to select and commit to courses of actions can easily lead to unbounded computations, further limiting its applicability to practical problems.

2.2.5 Hybrid Architectures

Hybrid architectures were created in an attempt to reconcile the characteristics of reactive and deliberative architectures [Wooldridge, 2002]; two very representative architectures of this kind are TouringMachines and InteRRaP. The TouringMachines architecture was created by Ferguson [Ferguson, 1995] to address a number of possibly conflicting issues, namely [Ferguson, 1995; Luck et al., 2004]:

- deal with unexpected events in the real world;
- deal with dynamism in the environment created by actions of other agents;
- pay attention to environmental changes;
- reason about temporal constraints in the context of resource-bounded computation; and

- reason about the impact of short-term actions on long-term goals.

Like Brooks's subsumption architecture described in Section 2.2.2, the TouringMachines architecture is organised in layers, but their interaction and operation are completely different. In TouringMagines, each layer represents a different level of abstraction, and can communicate with every other layer. Behaviour is generated by each of the three layers, with a *reactive layer* containing reactive behaviours needed for quick responses to a dynamic environment, a *planning layer* responsible for generating and executing plans to achieve long-term goals, and a *modelling layer* responsible for constructing a model of other agents and predicting their behaviour. These layers are illustrated in Figure 2.4.

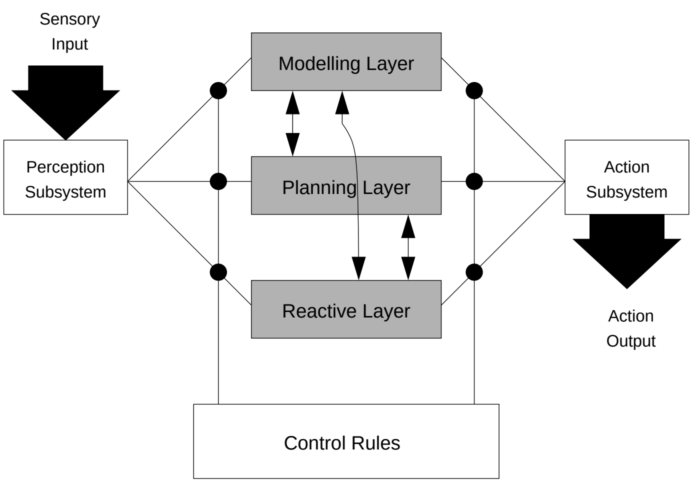


FIGURE 2.4: TouringMachine architecture [Ferguson, 1995].

InteRRaP, in turn, is an architecture developed by Müller [Müller et al., 1995], and consists of three vertically organised layers: behaviour-based, plan-based and cooperation. The behaviour-based layer performs reactive reasoning, and has rules concerning instant reactions and the execution of actions in the world. The plan-based layer is concerned with achieving longer term goals and *planning* to achieve them, whereas the cooperation layer deals with social behaviour. InteRRaP agents also have a knowledge base in the form of a hierarchical blackboard split into three layers corresponding to the reasoning layers. The lowest-level layer in the knowledge-base contains object-level beliefs about the world (*i.e.* the world model), while the middle layer contains information about the current goals and the adopted plans, and the topmost layer contains a model of the beliefs of other agents. These components, and their organisation are illustrated in Figure 2.5.

2.2.6 Discussion

As we have seen, research on agent architectures shifted emphasis initially from deliberative models to the reactive approach advocated by Brooks (among others) in his subsumption

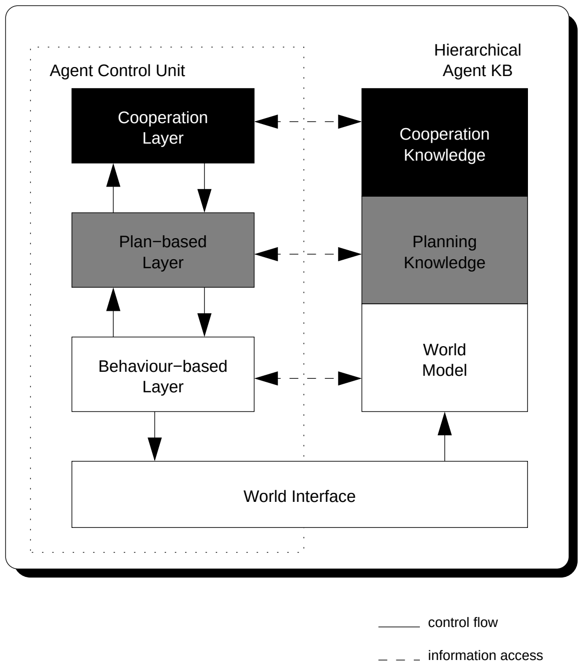


FIGURE 2.5: The InteRRaP architecture [Müller et al., 1995].

architecture, mainly because these initial deliberative approaches failed to result in practical systems. Although the reactive approach yielded initial successes in creating robots that can perform a number of low-level reasoning tasks, such as avoiding obstacles, reactive agents do not scale well for higher-level reasoning tasks, such as buying plane tickets. This limitation can be attributed to the difficulty in modelling and foreseeing the interactions among the large number of reactive rules required for these complex behaviours. Deliberative agents had a resurgence with the advent of the BDI model, which provides a folk psychology-based abstraction for describing and specifying agent systems, facilitating the design of more complex behaviours. Even though the BDI model assumes some kind of plan generation capability, the creators of PRS chose not to use it for reasons of efficiency, resulting in a system with characteristics of both reactive and deliberative architectures. This hybrid approach was also adopted by other systems such as InteRRaP and TouringMachines, trying to reconcile reactivity with deliberation.

2.3 Agent Languages

Given the number of architectures developed to realise agent systems, several researchers have sought to develop specific abstractions to allow designers to express these systems [Bordini et al., 2007; Dastani et al., 2004; Rao, 1996]. By specific abstractions, we refer to languages aimed at the description of agent systems. Therefore, in this section we review a number of significant agent languages created to facilitate the construction of agent systems. Although less extensive than agent architectures, the number of agent languages is large enough that a thorough listing of all existing languages is not feasible in the scope of this thesis. We therefore focus on languages that represent significant *milestones* in the chronology of agent languages. This section starts with the initial AGENT0 and its successor PLACA in Section 2.3.1, followed by AgentSpeak(L) in Section 2.3.2, and ending with 3APL in Section 2.3.3.

2.3.1 AGENT0 and PLACA

The first attempt at defining a *programming language* specifically using the notions of mental states and agency resulted in the AGENT0 programming language by Shoham [Shoham, 1993], who also coined the term *Agent Oriented Programming* (AOP). AOP is a programming model that requires a designer to create a set of transition rules defining how mental states change as a result of input received by an agent's sensors. AGENT0 agents comprise a set of beliefs, capabilities and obligations, and can communicate with other agents to either send information or to request that actions be carried out.

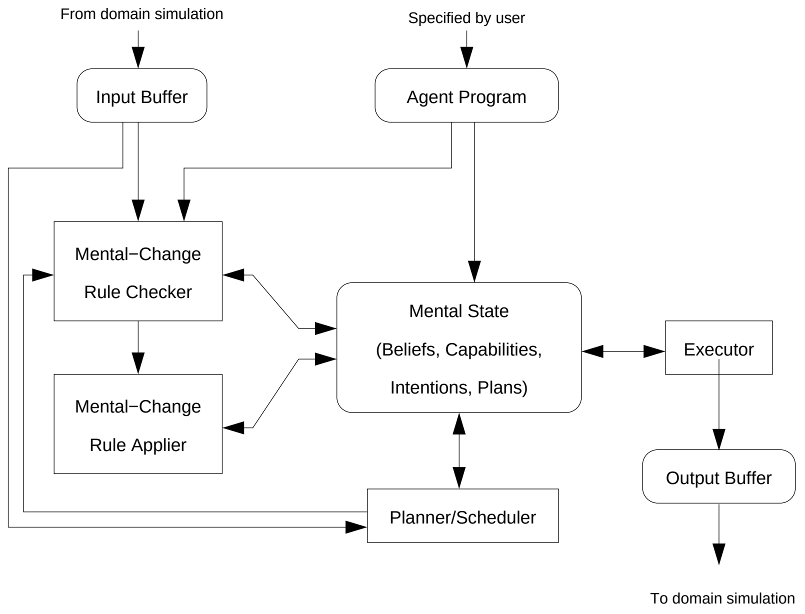


FIGURE 2.6: The PLACA interpreter [Thomas, 1995].

AGENT0 was succeeded by PLACA [Thomas, 1995][2], which extends AGENT0's expressivity with the addition of a planning component, and whose agents are defined in terms of an initial mental state and a list of mental-change rules. The initial state consists of a list of capabilities, and consistent lists of initial beliefs and intentions. From this initial state, a PLACA interpreter processes agent information together with perceptual data in order to generate actions. Thus, an agent starts its execution with an empty list of initial plans and, as it commits to intentions through its mental-change rules, new plans are created

[2]An acronym for PLAnning Communicating Agents.

by the planner and added to this list and eventually executed through the executor. This summarised execution cycle is illustrated in Figure 2.6.

2.3.2 AgentSpeak(L)

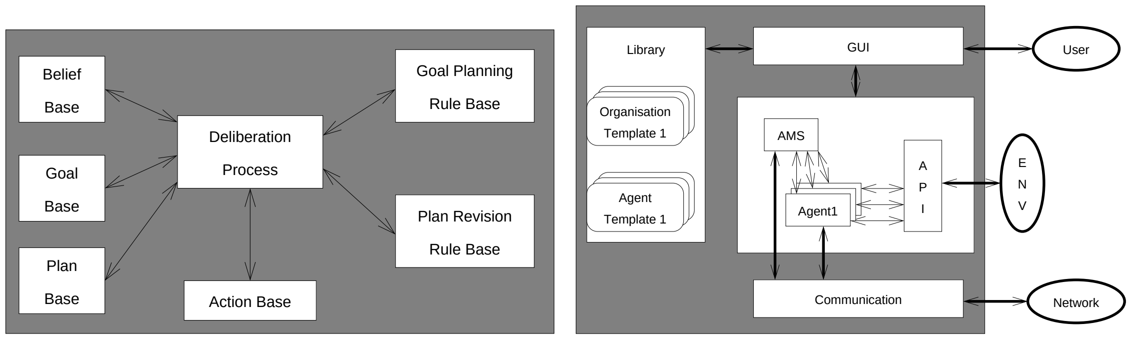
From a different perspective, the AgentSpeak(L) language [Rao, 1996] was created to bridge the gap between the theory behind BDI architectures and their implementation in systems like PRS [Georgeff and Lansky, 1987] and dMARS [d'Inverno et al., 2004]. This gap can be attributed to a number of factors [d'Inverno and Luck, 1998]: on the one hand, implementations are generally conceived in a simplified manner, resulting in the weakening of their theoretical underpinnings, while on the other hand, the logics used to build the theoretical basis seldom have a strong relationship with practical problems. To address this shortcoming, it is argued by Rao [Rao, 1996], it is necessary to provide a formal specification of the agents that are to be implemented. This formal specification can be achieved using a formally grounded agent language, and as a result, the AgentSpeak(L) language was created to provide a strong association to an underlying formal model. Rao claims that the semantics of AgentSpeak(L) should correspond to a formally defined version of the way in which the implementations of PRS and dMARS operate [Rao, 1996], thus bridging the gap between theory and concrete implementation.

AgentSpeak(L) is thus a programming language based on a restricted first order logic with BDI abstractions in which an agent is defined in terms of its initial beliefs and a plan library. Plans in a plan library are characterised first by an invocation condition indicating when plans can be adopted, and second by a context condition that represents the circumstance in which these plans are to be adopted. It hence avoids using the traditional modal logics of its theoretical origin. However, some of the BDI components are represented in a notably *implicit* way: in particular, the desires that a certain plan satisfy are not represented explicitly. As we shall see in more detail in Chapter 3, a plan invocation condition does not necessarily correspond to a particular desire an agent needs to satisfy, but rather a particular event in the world that may or may not have any connection with state an agent aims to achieve. In this way, an agent does not reason about its set of desires choosing plans to accomplish them. Rather, plans imply the goals they are expected to achieve in their invocation condition, so a desire is *implied* by the occurrence of some event expressed as this invocation condition.

2.3.3 3APL

3APL[3] is an agent programming language created to incorporate some of the concepts from agent logics [Hindriks et al., 1999] into an agent interpreter. The specification language of

[3]Pronounced "triple-a-p-l."



(a) 3APL Agent. (b) 3APL Platform.

Figure 2.7: General architecture of a 3APL agent and platform [Dastani et al., 2005].

3APL separates mental attitudes (data) and the deliberation process (programming instructions). Regarding the specification of mental states, 3APL agents need a definition of beliefs, goals, actions, plans and reasoning rules. Beliefs and goals are specified using logic formulas following the Prolog standard [Nilsson and Maluszynski., 1995], so that beliefs can either be static facts about the world or derivation rules, and goals specify desired world-states. This way of specifying goals makes 3APL one of the first agent languages to incorporate the notion of *declarative goals*, or explicitly specified target world-states. *Capabilities* define the preconditions and effects of actions in a format similar to STRIPS, which can then be used in plans in the agent's plan base. Besides these data structures, a 3APL agent contains two rule-bases, one called *goal planning rules* (or PG-Rules) for "generating" plans (though not in the planning sense) to achieve goals and another called *plan revision rules* (or PR-Rules) to revise plans from the plan base. These elements are illustrated in Figure 2.7(a). PG-Rules are in a sense analogous to the invocation conditions of AgentSpeak(L) plans, while PR-Rules provide a mechanism to modify plans that fail to achieve their designated goals.

In addition to these components, 3APL includes a meta-language that allows customisations to the process of selecting which plans to execute, which goals to pursue and which rules to apply. In principle, this meta language allows a designer to use constructs similar to those available for plans in the plan base to redefine the way in which the deliberation process operates.

In order to program *multiagent* systems, Dastani *et al.* [Dastani et al., 2005] in a further development presuppose two programming languages: one to implement single agent systems and another to implement multiagent aspects. Here, the function of the multiagent language is fulfilled by a Java-based platform, in which a shared environment is programmed directly in Java. The multiagent platform used by 3APL, illustrated in Figure 2.7(b), allows the deployment of multiple agents, whose management is controlled by an *agent management system* (AMS) that includes a directory service to locate agents.

2.3.4 Discussion

Throughout the evolution of agent languages, emphasis has been given to a strong association of the language with some formal semantics, which is the case with AGENT0, AgentSpeak(L) and 3APL. Each of these languages focused on particular properties, with AGENT0 focusing on simply describing a distributed system using mentalistic notions, AgentSpeak(L) focusing on correspondence with a popular implementation of an agent interpreter, and 3APL focusing on realising a particular BDI logic. It is interesting to note that AGENT0's successor, PLACA, included the notion of planning as an essential capability of the agents created using that language, which other, more recent, languages have omitted.

AGENT0 and PLACA are admittedly initial approaches to agent programming, and as such they did not aim to be definitive solutions for agent programming. AgentSpeak(L), conversely, follows the tradition of pragmatic agent implementations, but this also reflects on its simple mechanism for action selection geared towards execution efficiency. Unlike both PLACA and AgentSpeak(L), 3APL focuses on mixing logical aspects of agent languages with bindings to Java to provide a more programmer-oriented language, but like AgentSpeak(L), it relies on predefined plans to solve problems at runtime. Unlike the previous languages, however, 3APL contains a meta-language that allows aspects of the reasoning process to be manipulated at runtime, which is a concept we will explore in Section 2.4.

2.4 Meta Level Control

As we have seen, 3APL was one of the first languages to incorporate an auxiliary language to manipulate the reasoning process at runtime. This manipulation follows the idea that in order to act effectively in any complex environment, autonomous agents must have control over their internal state and behaviour [Jennings, 2000]. To exercise this control, an agent needs some means of reasoning about its internal state, often in a process known as meta-level reasoning (or meta-reasoning). This is higher level reasoning about the reasoning process itself, and in agent systems it is commonly used in enforcing rationality in the *choice* of goals achieved or actions performed by an agent, ensuring that the agent behaves as effectively and efficiently as possible. Conversely, traditional agent architectures often reason only at the *object level*; that is, their reasoning is limited to collecting perceptions from the environment (or *ground level*) and acting upon this environment. We illustrate the differences between these levels of reasoning in Figure 2.8.

Through meta-reasoning, an agent is able to explicitly consider goals before committing to them, and consider courses of action before executing plans, in contrast to simply reacting to events in the environment. Meta-reasoning is typically considered in two distinct ways: as a behaviour optimisation process, or as a behaviour scheduling mechanism. First, in

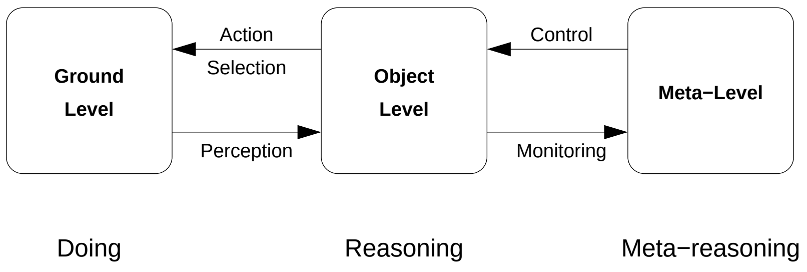


Figure 2.8: Reasoning and meta-reasoning [Cox and Raja, 2008].

the context of optimising agent behaviour, meta-reasoning improves behaviour over time by observing the results of previously selected courses of action and adjusting the selection mechanism. This optimisation process is only possible when there is some objective method of assessing action effectiveness, so it is mostly suitable to domains in which there is some natural way of determining whether a certain behaviour is good or bad. Alternatively, meta-reasoning has been used in architectures based on continuous planning, which consists of a planning process that can arrive at a plan quickly, but can improve this plan given more time. In these architectures [Ambros-Ingerson and Steel, 1988], meta-reasoning is used to decide when enough planning has been performed and initiate action execution. From a BDI perspective, this latter type of meta-reasoning can be used to select an intention to execute from among a set of concurrent intentions. For both applications of meta-reasoning, there must exist an objective criterion to assess the quality of an agent's reasoning, such as resource consumption in the environment or the failure rate of selected plans.

In this section, we examine research on models of meta-level control comprising both behaviour optimisation and scheduling architectures. We end with a discussion on how meta-reasoning can be incorporated into an existing agent architecture and indicate possible abstractions for this type of reasoning.

2.4.1 Meta-level control in deliberative agents

Raja and Lesser [Raja and Lesser, 2004] describe an approach for meta-level reasoning with bounded computational overhead, which can be dynamically constructed and improved through reinforcement learning. This approach is intended to improve agent performance in complex domains where there is a need for dynamic decision-making about tasks. Here, meta-level control is described in terms of meta-level actions that can be used whenever a new task arrives. Among the available options for meta-level tasks is the execution of two types of scheduler, a simple scheduler, that uses pre-computed information to arrive at an acceptable execution schedule, and a detailed scheduler, which takes into consideration several optimisation metrics and calculates rewards in soft real-time. A *meta-level controller* (MLC) selects one such action, which can do one of the following:

- drop the task;
- use the simple scheduler on the new task;
- use the detailed scheduler on the new task;
- use the detailed scheduler on the new task and all tasks including partially executed ones;
- add the new task to the agenda; or
- gather more information about the task and try to decide again.

During this decision process, the MLC compares the deadline of the new task with the available time and spends this time in trying to maximise the utility obtained and satisfy the deadline. This decision of which meta-level action to take is made using a decision tree which has information regarding the expected utility of each meta-level action given a certain agent state. The learning process described by Raja and Lesser uses a domain-independent abstract representation of the agent state to refine the decision tree used by the MLC.

It is unclear how the abstract representation used by Raja and Lesser can be used to design meta-level strategies, and how the utility of meta-level actions relate to the utility of the action-directed plans being scheduled for execution. This latter problem is of great concern, since it is generally not the meta-level actions of an agent that cause utility gains, but rather the actions an agent execute upon the environment.

2.4.2 Thangarajah's Detection of Goal Interactions

Given the possibility that goals being pursued in parallel may result in the execution of plans which may interact, Thangarajah *et al.* [Thangarajah et al., 2003a,b] state that a rational agent should be able to detect these interactions and avoid negative ones while exploiting positive ones. Negative interactions occur when the satisfaction of one goal prevents the achievement of another goal. Positive interactions may occur when two goals have overlapping dependencies or subgoals, and a rational agent can achieve these dependencies only once in the process of achieving both goals.

Positive interactions are exploited by *plan merging* during scheduling. This means that if two goals have the same subgoal, only one plan to achieve the subgoal will be scheduled for execution, so it will be achieved only once for both goals. In order to reason about plan and goal interactions, Thangarajah *et al.* [Thangarajah et al., 2003b] developed a representation of goals and plans called *Goal-Plan Tree* (GPT) that structures goals as nodes having all possible plans to achieve it as children, while each subgoal included in these plans is, recursively, a node down this tree.

Plan templates (or plan types) are described in terms of an identifier, pre-conditions, in-conditions, post-conditions and a plan body. Pre-conditions specify when a plan can begin execution, analogously to AgentSpeak context conditions. In-conditions are conditions that must be true throughout the execution of a plan, otherwise the plan fails. Post-conditions or effects are the conditions that will hold after the plan has been executed to achieve a certain goal. Finally, a plan body contains actions and subgoals. Goal types are represented in a similar way to plans, *i.e.* with an identifier, and in-condition, effects and plans. Here, effects are the success condition sought by the goal, and the plans are an enumeration of the possible plan-types that can satisfy this goal. It is important to note that plans are explicitly associated with goals in this representation, and a distinction is made between plan effects and goal effects, so that the effects of a plan that are not the effects of an associated goal are viewed as side-effects.

The effects of goals are classified as either definite or potential. Definite effects occur in all possible paths to achieving a goal, whereas potential effects are those that may occur in some paths but not others. This information is summarised in *effect summaries*, which are built by propagating the effects of a tree of potential paths to achieving a goal. Effect summaries are used to facilitate the process of recognising and exploiting positive goal interactions. By analysing effect summaries of concurrent plan structures, it is possible to identify plans that achieve the same goal and eliminate redundant plans, or to merge compatible plans to minimise redundant steps.

This work by Thangarajah *et al.* shows an interesting application of meta-reasoning to improving the efficiency of an agent at runtime. Importantly, this approach underlines the importance of being able to deduce the declarative effects of plans in traditional agent architectures in order to detect when plans may interfere with each other. In this particular work, this information is supplied in the form of effect summaries, which is an approach that can be leveraged in this thesis.

2.4.3 Pokahr's Goal Deliberation

Though most agent languages allow for the execution of multiple plans in parallel, the underlying systems ignore the issues relating to actually executing conflicting plans in parallel, as there is no framework for deciding how goals interact and how to choose them. The aspect of goal deliberation dealt with here is how to deliberate on possibly conflicting goals to decide which ones are to be pursued. Since Pokahr *et al.* [Pokahr et al., 2005a] aim to integrate this type of deliberation into an agent architecture, the underlying infrastructure has to provide a clear interface for goal handling, conflict resolution and exploitation of positive interactions. Moreover, meta-level strategies regarding goal deliberation have to take into account at least three issues:

- the important influence factors that can be used to drive the decision process;

- when and how often to deliberate about goals; and
- about which goal sets to deliberate.

In order for goal deliberation to take place, an explicit representation of goals is necessary, *i.e.* goals must have some sort of declarative representation. This framework is implemented in an architecture that uses a modified version of the traditional BDI interpreter cycle, which includes a set of meta-actions invoked as needed. In this particular architecture, goals are defined as having three properties, or conditions, which are monitored throughout their lifecycle: a creation condition, a context condition (which when not true entails the suspension of the associated goal), and a drop condition. This goal representation follows the ideas for goal modelling from Braubach *et al.* [Braubach et al., 2004].

The influence factors used to drive the decision process considered by Pokahr *et al.* are cardinalities and inhibition arcs. Cardinalities restrict the number of active goals of a specific type (each goal description constituting a type), while inhibition arcs are used to declare negative interactions between two goals. Deliberation occurs on demand, as a result of new goals being adopted or existing goals needing to be suspended, and the goal set considered for deliberation is the subset of the active goals related to the goal that triggered deliberation.

Pokahr's efforts underline the importance of a declarative representation for goals in performing meta-level reasoning. Thus, without a representation of the expected results of plans and actions, an agent cannot prioritise goals unless a designer introduces arbitrary reward values for specific plans.

2.4.4 VHD++

De Sevin and Thalmann [de Sevin and Thalmann, 2005a,b] describe a model of motivated action selection for simulated humans that uses multiple motivations (*e.g.* hunger, thirst) and their intensities to provide a hierarchy of preferences for the triggering of goal adoption rules. This additional level of abstraction between raw sensor data and goal adoption using motivations is called a *hierarchical classifier system*. It is intended to reduce the search space analysed in the action selection process by the use of weighted rules corresponding to motivations that lead certain goals to be prioritised. In turn this can improve the modelling of complex systems with large numbers of rules for the selection of goals and actions.

The evaluation of motivations relies on two threshold values that define three *zones* of motivation: a *comfort zone* below the first threshold; a *tolerance zone* between the threshold values; and a *danger zone* beyond the second threshold. This evaluation is non-linear in that intensity levels increase slower in the tolerance zone, with no action being taken by the agent, while if the intensity lies in the danger zone its evaluation is amplified to increase the odds of action towards its satisfaction. These zones of motivation and an example of how motivational intensity might increase through them is illustrated in Figure 2.9.

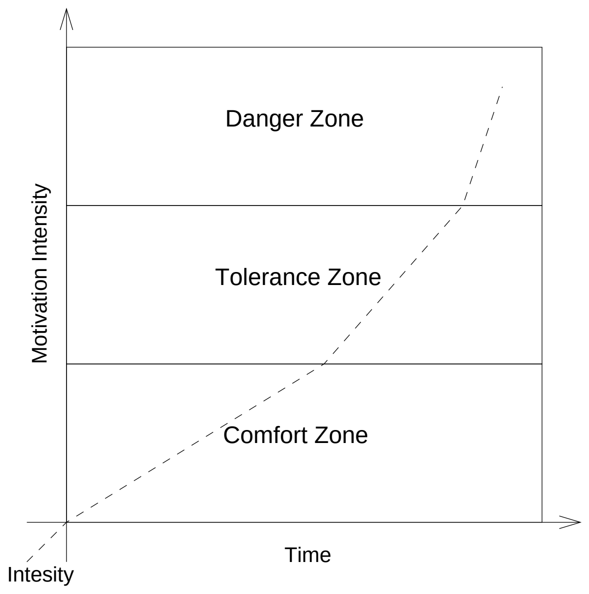


FIGURE 2.9: The three zones of motivation [de Sevin and Thalmann, 2005a].

A complete agent architecture, including this model of motivated control, is implemented in VHD++ [de Sevin and Thalmann, 2005b], which is used in various simulation scenarios involving human needs, modelled as motivations such as hunger, thirst and rest. Here, the motivation model is able to modify an agent's internal state, leading it to adopt certain behaviours for the achievement of long-term goals, and also causing it to carry out actions directly, in order to react to some immediate need or opportunity.

The model also includes a learning module to allow motivation thresholds to be adjusted to optimise behaviours, as well as a social component to allow the agent to infer the motivations of other agents and respond to them. Since this method of motivated behaviour is used directly for action selection, it includes a simulation of hysteresis[4] to prevent abrupt changes in behaviour and to allow for the agent to persist in certain behaviours before considering alternatives. While this particular mechanism is interesting if the only mechanism for action selection is the motivational model, it is unnecessary in a BDI architecture, since intentions provide the means for persistence of plan-long behaviours.

2.4.5 Motivational quantities and organisations

In order to allow agents to weight the rewards of certain behaviours and actions, Wagner and Lesser [Wagner and Lesser, 2000] define a simplified model of *motivational quantities* (MQ),

[4]Hysteresis is the lagging of an effect behind its cause; in this case, the motivations start to change slightly after the events that cause their change.

inspired by more traditional views of motivational dynamics in agents [Decker, 1998]. In this model, interacting agents belong to distinct organisations, and information regarding the relationships between these organisations is used by agents to reason about the rewards of providing or utilising services from agents across organisations. Agents with an augmented understanding of the organisational context affect the control of a multiagent system in two primary ways:

- information about the structure of an organisation narrows the scope of an agent's horizon of interaction, preventing a combinatorial explosion of interaction possibilities; and
- information about the value of interactions and actions affects the choice of actions and goals of an agent.

When choosing which task to carry out next, socially situated agents using the MQ model consider three classes of task:

- local concerns with no value to others;
- tasks that others wish the agent to perform; and
- tasks that others may perform for the agent.

Agents have a set of MQs that they accumulate and exchange with other agents, and which may be based either on common intrinsic MQs or on MQs acquired dynamically. Task execution produces MQs of one or more types, representing the benefit of performing a given task. Conversely, some tasks may cause the loss of MQs, representing a negative outcome to the agent. In addition, organisational information affects an agent's representation of MQ production and loss, steering the agent to choosing actions that are valuable not only for itself but for the organisations in which it is a participant.

2.4.6 Other

In addition to the control systems explored in this section, a number of other, less recent, architectures are also reviewed, and we mention them briefly in this section.

First, the Alarms architecture allows agents to generate goals asynchronously to focus resources on the accomplishment of important goals [Norman and Long, 1995]. This process of asynchronous goal generation entails that new goals can be generated before current ones are accomplished, so that it is possible for an agent to adopt more goals than it can effectively work on at the same time. Moreover, adopted goals require processing resources for scheduling and planning, and since any agent has a limit on its processing resources regardless of its efficiency, there must be an upper bound for the number of goals it can pursue simultaneously. When this bound is exceeded, the agent will no longer function effectively.

Next, the *Will* architecture was created by Moffat and Frijda [Moffat and Frijda, 1995] as a means to integrate multiple AI techniques as isolated components within a complete mind using the simplest possible working model. A Will agent comprises a set of components that operate asynchronously with no awareness of each other, organised around a *memory* component upon which they read and write information, similarly to a Blackboard architecture [Hayes-Roth, 1985]. These components are responsible for perceiving the environment, reacting to events, planning for goals and executing the actions selected by the agent in an asynchronous manner.

More recently, the Abbot architecture proposed by Cañamero [Cañamero, 1997] is based on motivational and emotional states modelled as *Society of Mind* (SoM) agents [Minsky, 1986] that influence each other trying roughly to emulate the operation of the emotional and basic hormonal system. The Abbot architecture was tested in an abstract world in which some agents interact, aiming to satisfy their urges.

2.4.7 Discussion

While the systems and models considered above provide some valuable solutions to the problem of meta-level control, ranging from simple reaction rules to more refined behaviour selection processes, they have some shortcomings that have prevented their mainstream adoption in agent development.

The first three efforts reviewed in this section focus on abstract models of meta-reasoning: Raja and Lesser [Raja and Lesser, 2004] (in Section 2.4.1) use meta-reasoning to allow the scheduling of existing tasks based on a pre-defined preference relation; Pokahr *et al.* [Pokahr et al., 2005a] (in Section 2.4.3) and Thangarajah *et al.* [Thangarajah et al., 2003b] (in Section 2.4.2), rely on a technique that summarises the effects of plans considered for adoption and analyse positive and negative interactions to avoid conflicts and maximise opportunities. These efforts improve agent efficiency by focusing on specific areas of the reasoning process to optimise. However, such optimisations rely on detailed knowledge of their underlying architectures [Pokahr et al., 2005a; Thangarajah et al., 2003b], or on some abstract notion of utility to allow prioritisation [Raja and Lesser, 2004], and all use a single, static strategy to improve agent efficiency.

The efforts we reviewed afterwards focus on a particular abstraction for meta-level reasoning, but in the context of a customised agent architecture. VHD++ and MQ lack a complete agent architecture and language, and are therefore *ad hoc* methods for building agents. In two of these models some of the features of previous agent models (namely BDI) are replicated in an *ad hoc* way. The model of de Sevin and Thalmann (in Section 2.4.4) allows a degree of *hysteresis* to prevent switching too fast between behaviours, which is equivalent to the notion of commitment in BDI, but it is not clear how much hysteresis any given behaviour or agent requires in order to achieve an acceptable degree of commitment.

It is clear that the Abbot, Will and Alarms architectures (outlined in Section 2.4.6) were designed as testbeds for specific notions of motivation, hence they lack the tooling for the generic development of agents. Abbot agents use motivations in a Society of Mind context, in which behaviours are expected to emerge from the interaction of multiple dissociated components, but emerging behaviours are difficult to model in a predictable way for complex scenarios. Will agents use a BDI-like approach in dividing the architecture into multiple mental abstractions, but they lack the tools and a language to allow generic agent development. Finally, the Alarms architecture demonstrates a series of advantages that motivation-based meta-level control can provide, but its abstractions for motivations and their relation to the choice of actions is somewhat confusing.

More generally, it appears that meta-level control is more easily applied to domains in which agent behaviour can be easily evaluated as being effective through some existing criterion. When there is no such criterion, it is necessary to create some way in which to evaluate an agent's mental state to provide guidance for plan selection. As a consequence, most of the meta-reasoning architectures discussed in this section use the abstraction of motivation to allow evaluation criteria to be specified regardless of the domain. In this context, it seems that motivations can provide an elegant solution for modelling meta-level control in an agent system. Motivations are regarded by many as *an orientation towards certain classes of goals* [Luck and d'Inverno, 1998; Morignot and Hayes-Roth, 1996], and are used in various theories of animal behaviour to explain why animals behave in certain patterns [Balkenius, 1993]. Motivations are also used to provide a mechanism of *lateral inhibition*, suppressing unnecessary behaviours in order to give priority to more urgent ones. Motivations have not been *explicitly* advocated as a mechanism for meta-level control, but given the above review, motivations appear to be a natural mechanism for meta-level control, as they can be used analogously to such control mechanisms.

2.5 Multiagent Systems

Considering our research objective of a planning-capable agent architecture that is able to operate in a society, and having already reviewed efforts in agent architectures, languages and meta-reasoning, we now focus on the societal aspects of agent systems. In this section, we review a number of important efforts in the field of multiagent systems that are critical to the creation of effective agent societies. However, as before, due to the immense amount of work in this area we must limit our review to a representative sample of work. We start in Section 2.5.1 by considering a basic element in multiagent systems, which is the communication language required for agents to interoperate, and since our focus is on being able to deal with situations not envisaged at design time, we review aspects of multiagent planning in Section 2.5.2. Finally, in order to provide a mechanism to ensure that undesirable behaviours do not arise in a system of autonomous flexible agents, we review the application of norms to agent systems in Section 2.5.3.

2.5.1 Communication Languages

Considering the interactive nature of agent systems, communication is naturally an important concern. Regardless of the similar origins between distributed systems and agent systems, communication between agents occurs at a higher level, requiring some sort of higher level abstraction surrounding communication. Unlike lower-level interprocess communication in which messages either provide irrefutable information and undeniable commands to execute code, agents may accept or reject the validity of new information, as well as deny requests for action.

Agent communication generally follows the theory of *speech acts* proposed by Austin [Austin, 1962]. Speech act theory states that certain utterances can be considered *actions* in some contexts, and Austin calls this class of utterances *speech acts*. Speech acts have three distinct aspects: the *locutionary act*, which is the act of vocalising a certain message; the *illocutionary act*, which is the actual action performed as a result of the locutionary act; and the *perlocution*, which is the effect of the illocutionary act. Speech act theory was later extended by Searle [Searle, 1969], identifying several types of conditions for speech acts to be performed, and classifying a number of possible speech act types, such as commitments to the truth of information, requests and commitments for action, among others. In the following sections, we describe two concrete speech act-based agent communication languages, representing an initial effort in adapting speech-act theory to agent communication and a standardised evolution of this first effort, respectively.

The Knowledge Query and Manipulation Language (KQML) [Finin et al., 1994] and Knowledge Interchange Format (KIF) [Genesereth and Fikes, 1992] are two languages developed under a DARPA-funded initiative to provide communication protocols for autonomous information systems. KIF is not a communication language in itself, but rather a knowledge representation format intended to be used as the content of messages in other languages, such as KQML.

KQML is the message protocol counterpart to KIF, consisting of speech act-based messages. Each message, therefore, has a *performative* (that is an illocutionary force) determining the general purpose of the message and parameters. KQML includes performatives such as: *ask* to request some information; *achieve*, to achieve a certain world-state in the environment;

```
(inform
    :sender   agent1
    :receiver agent2
    :content (price good2 150)
    :language sl
    :ontology hpl-auction
)
```

Table 2.1: Example of FIPA ACL message.

tell, to inform another agent of a certain belief; and several others. It became very popular among the agent community, but a lack of a formal semantics meant that it was hard to ensure interoperability among agents communicating using different implementations of KQML. This lack of semantics compounded with the existence of performatives of dubious utility led to the creation of a more formally defined language by the Foundation for Intelligent Physical Agents.

In an attempt to create definite standards for agents, the Foundation for Intelligent Physical Agents (FIPA) proposed its own version of an agent communication language (ACL) [Foundation for Intelligent Physical Agents, 2000], which was significantly influenced by KQML and with messages naturally similar to KQML, as illustrated in Table 2.1.

While KQML has a large number of loosely defined performatives, some of which can be interpreted in many ways, and leading to possible problems in interoperation, the FIPA standard contains a smaller number of performatives with a very precise meaning. Every performative in FIPA is defined in terms of two basic performatives: *request*, used to request that another agent perform some action; and *inform*, which means that an agent wants another agent to believe in some information.

2.5.2 Multiagent Planning

In order to solve problems in a distributed manner, multiple agents need both group coherence and competence [Durfee, 2001]. Coherence relates to agents deciding to work together, while competence relates to agents knowing how to work together well. Perhaps the best way of solving distributed problems is through planning, though planning in a distributed fashion requires another problem to be solved, that of planning how to work together. This activity of *planning to plan* involves decomposing problems into subproblems, allocating these subproblems, exchanging the obtained solutions and synthesising overall solutions [desJardins et al., 1999]. We consider two distinct phases of multiagent planning: plan generation in Section 2.5.2.1 and distributed plan execution in Section 2.5.2.2. We then examine two important distributed planning algorithms in Sections 2.5.2.3 and 2.5.2.4.

2.5.2.1 Distributed plan generation

Distributed planning can be categorised according to whether distribution occurs at the execution or planning stages, or both [Durfee and Lesser, 1991]. There are three possible combinations of where the distributed part of planning occurs. When *centralised planning* is employed to produce *distributed plans* for parallel execution, the problem is for a coordinator agent to break a plan into separate threads, possibly adding synchronisation actions throughout the plan in order to ensure that dependency constraints are observed during plan execution. In order to distribute execution of a centralised plan, Durfee [Durfee, 2001] proposes the following steps:

- create a plan using traditional planning, possibly biasing the planner to favour parallel actions;
- decompose the ensuing plans trying to minimise ordering constraints between subplans;
- insert synchronisation actions;
- allocate subplans to executing agents; and
- execute the subplans.

However, the overhead incurred by the communication required for subplan synchronisation must be taken into account when deciding whether or not to distribute a centralised plan; that is, there exists a minimum subplan size below which parallelisation is not worthwhile.

Distributed planning for centralised plans is associated with cooperative planning, in which a complex planning problem is distributed among different specialised agents which contribute parts of the solution to an overarching plan, in a very similar fashion to that of result sharing.

Finally, *distributed planning for distributed plans* is the mode of planning most representative of problem solving in multiagent societies [Durfee, 2001]. In this case, a plan representation for the whole plan is not required to exist at all, as long as the participant agents are not in conflict during the planning and execution tasks.

2.5.2.2 Distributed plan execution

In addition to the planning process, distributed *execution* must be carried out in a coordinated manner to ensure the originally intended sequence of actions occurs as planned. That is, agents must somehow ensure that: agents execute their assigned actions at the appropriate times; the actions of one agent do not jeopardise the actions of another agent working for the same goal; and agents do not compete for control of critical resources. In turn, coordination might be interleaved between continuous planning and execution, such as in *partial global planning* [Cox et al., 2005; Durfee, 1988], or might be guaranteed either before or after the planning process is carried out. Moreover, when dealing with self-interested agents, agents may be required to negotiate during distributed planning in order to resolve conflicts. Failure to do so can lead to system collapse, so negotiation mechanisms that facilitate the resolution of critical conflicts are an important component of a distributed planning strategy.

One way of ensuring coordination after plans are created is through *contingency planning* [Meuleau and Smith, 2003]. In this approach, an agent not only plans to satisfy the specified problem, it also creates alternative plans to be resorted to in response to contingencies occurring at execution time. Clearly, this entails more complex plans, as well as an overhead to the execution and coordination process, which must now consider the possible threads of plan execution. Aside from contingency planning, agents can monitor execution progress and replan if problems arise. Nevertheless, too much replanning can become a liability, in which case plan repair could be an advantageous approach [desJardins et al., 1999].

Pre-planning coordination, on the other hand, involves defining a set of constraints that are enforced by the agents in the society during their individual planning processes [desJardins et al., 1999]. If the appropriate set of constraints is defined, agents can theoretically work on any part of the problem, since conflicts can be avoided by carefully abiding by these constraints. Another way of viewing these constraints is as *social laws*, which encode prohibitions against particular choices of actions in particular contexts. This in turn implies the design of combinations of laws that curtail undesirable states, yet are flexible enough to allow for the desired states to arise from the agents, and we examine such normative approaches in Section 2.5.3.

2.5.2.3 Partial Global Planning

Partial Global Planning (PGP) [Durfee and Lesser, 1990] is a distributed planning framework that adopts a strategy where coordination is a matter of explicitly planning cooperative interactions [Durfee and Lesser, 1991]. In this approach, all agents maintain a partial representation of the global plan, and no agent is assumed to be able to see the entirety of the global plan. Here plans detail an agent's problem-solving strategy and its expectation about the actions of neighbouring agents, and although agents attempt to follow their partial plan as closely as possible, they can also make changes to their plan or propose changes to the plans of other agents. The PGP framework integrates organisational principles by introducing two types of organisation: the first specifies the long-term problem-solving roles and responsibilities of agents (*i.e.* a plan of actions); the second, or metalevel organisation, gives agents a framework for deciding how to solve coordination problems (*i.e.* a plan of communication). Agents are expected to exchange information about the state of their plans to a certain extent, sharing only high-level information deemed relevant to the agents being informed. They can also perform task-sharing by proposing (and counterproposing) the transfer of a part of their local plans to other agents that might be underutilised.

2.5.2.4 Generalised Partial Global Planning

Generalised Partial Global Planning (GPGP) [Lesser et al., 1998] shares with PGP the idea that agents construct their own local view of the tasks they intend to pursue and the relationships among them. This local view can then be augmented with information from other agents, allowing agents to create a partial view of the global plan. The generalised framework extends PGP by including individual *coordination* mechanisms used in the creation of such partial views, detecting relations between task structures and ensuring coherent and coordinated behaviour by making commitments to other agents. In turn, these commitments are used by a domain-independent scheduler included in GPGP to create a schedule of activities for the agent to follow.

GPGP incorporates a representation of task structures from the TAEMS [Wagner et al., 1997] framework to drive the coordination mechanisms, including information about:

- top-level goals an agent intends to achieve;
- one or more of the ways in which these goals could be achieved;
- a precise quantitative specification of the degree of achievement of goals; and
- task relationships indicating how tasks contribute to the achievement of goals.

GPGP uses the basic TAEMS task structure representation and adds the partial representation of the task structures held by other agents as well as local and non-local commitments to task achievement. Moreover, the quantitative definition of the degree of achievement for goals and tasks indicates that GPGP deals with *worth-oriented* domains rather than the boolean representation of achievement often used by planning algorithms.

2.5.3 Norms

We have seen that agents in a society might be able to generate new plans to address unforeseen situations, but agents capable of generating new plans at runtime might lead to undesirable behaviour, requiring some means to ensure that new behaviours abide by certain constraints. Moreover, in open dynamic societies, agents are required to work with others that do not necessarily have the same set of objectives. If left unchecked, self-interested agents will try to accomplish their individual goals without regard for others. Thus, in order to minimise conflict between self-interested agents, systems of prescriptive *norms* can be used to specify permissions, prohibitions and responsibilities within a system. In this section, we examine a number of views of normative systems, but unlike previous sections we do not focus on specific systems or techniques, since practical agent architectures have generally not considered or facilitated the possibility to include reasoning about norms.

Norms can be explicitly represented and reasoned about by agents to which they apply. Though the precise semantics of norms varies through different research efforts, Lopez y Lopez [Lopez y Lopez, 2003] identifies six different perspectives.

- *regulation of human societies*, where research focuses mainly on the sociological aspects of human normative bodies;
- *patterns of behaviour*, used to foster coherent group behaviour without the need for explicit planning of coordination actions;
- *constraints on actions*, used to specify permitted and forbidden actions [Shoham and Tennenholtz, 1995];
- *social commitments*, where norms express obligations among agents;
- *mental states*, focusing on the influence exerted by norms upon the adoption of goals by an agent; and
- *norm modelling*, focusing on the definition of the concept of norms and the specification of models of norms.

2.5.3.1 Levels of normative behaviour

Dignum [Dignum, 1999] argues that even agents said to be autonomous are assumed to obey standard protocols, so are predictable in some ways, implying some level of knowledge of the internal mechanisms of these agents. Here, predictability is the result of a set of conventions hard-wired into an agent, undermining the actual autonomy of the agent and consequently its ability to react to a dynamic environment. Dignum states that in order for an agent to be truly autonomous, it must be able to reason about the norms to which it should abide, and occasionally violate them if they are in conflict among themselves or with the agent's private goals. Thus, Dignum [Dignum, 1999] defines logical modalities for obligations and permissions (*i.e.* a deontic logic), distinguishing between three levels at which agent behaviour is influenced by these norms:

- the conventions level;
- the contract level; and
- the private level.

This division into levels allows the definition of rules for the different social interactions of an agent. The conventions level covers obligations that constitute a *default background* against which agents interact. These obligations hold under normal circumstances unless higher priority concerns intervene. Norms in the conventions level are generally fixed when the system is initialised and represent general rules for agents in a system to follow (termed *prima facie* norms), such as *agents should not overprice their goods.* Modalities at this level specify obligations, prohibitions and permissions that hold between a given agent in relation to an undefined or abstract counterpart (*i.e.* the agent society). Since there is no specific counterpart towards which the norm is directed, it is assumed that agents follow the rule either due to a commonsense benefit, or due to agents in charge of enforcing conventions.

The contract level covers commitments between agents, in the form of either *directed obligations* or *authorisations.* Contracts express the expectation of one agent towards another as well as the conditions for these contracts to hold and the consequences of failing to fulfill them. Directed obligations express a commitment from one agent to another that either a world-state will hold or an action will be executed. Authorisations express the justification of an agent to perform an action involving another agent; for example, if an agent is to demand payment from another (implying that the latter agent is obliged to pay), it must be authorised to do so.

The private level is used to translate the influences received from the other levels into something that directs an agent's future behaviour. For example, in a BDI setting, external influences and their conditions can be translated into conditional desires for an agent.

2.5.3.2 Norm types

Following the view of Dignum [Dignum, 1999] regarding the requirements of norms for autonomous agents, Lopez y Lopez and Luck [Lopez y Lopez and Luck, 2003; Lopez y Lopez et al., 2004] define a formal model of norms whose constructs are reasoned about by autonomous agents. In this model, norms are *prescriptive* in that they specify how agents should behave, and *social* as they are used in situations where multiple agents might come into conflict. Moreover, given the possibility that norms might conflict with an agent's individual goals and that punishments are defined for non-compliance, norms also represent a form of *social pressure* upon the agent.

Depending on their purpose, norms are classified as obligations, prohibitions, social commitments and social codes [Lopez y Lopez and Luck, 2003], where:

- obligations and prohibitions are norms aimed at ensuring coordination among agents in a society, non-compliance of obligations entails punishment, and the manifestation of behaviours targeted by prohibitions leads to punishment;

- social commitments are norms created as a result of agreements or negotiations between a group of agents in order to force them to comply with the agreement or settlement; and
- social codes are norms whose compliance is seen as an end in itself, as it is understood that these are principles accepted in a given society.

2.5.3.3 Norms for autonomous agents

Because norms in a given system are rarely isolated from each other, in Lopez y Lopez and Luck's model, systems of norms are created to ensure that agents comply with whole sets of norms rather than choosing individual norms with which to comply. Systems of norms can also be used to maintain consistency among constituent norms. The association of multiple norms can be attained by relating the activation of a given norm to the violation (or fulfilment) of another through *activation triggers.* Such triggers can be based on agents failing to comply with a norm (*i.e. non-compliance*), in which case a secondary norm is activated to punish the non-compliant agent. Alternatively, agents can be encouraged to comply with certain norms if other norms are created to trigger rewards to compliant agents. These triggers may serve the purpose of either punishing norm violators or rewarding norm followers. In case a violator requires punishment for a transgression, an *enforcer* norm might be activated following the transgression. Alternatively, achievement of a prescriptive goal might trigger a *reward* norm so that the compliant agent will be rewarded. Finally, norms may be used to provide for the evolution of the normative system itself. In this context, *legislation* norms are used to permit actions to issue new norms or abolish existing ones.

Since normative systems are maintained within the society employing them through delegation of punishment, reward and legislative goals, the effect of these systems upon prospective members of these societies can also be reasoned about by autonomous agents. When deciding whether to voluntarily join or leave a society regulated by norms, Lopez y Lopez *et al.* [Lopez y Lopez et al., 2004] advocate that an autonomous agent must have an additional set of characteristics to include ways of reasoning about the advantages and disadvantages of abiding by the norms, thus leading to the possibility of norm infringement. Transgression of norms might occur for three main reasons [Lopez y Lopez et al., 2004]:

- individual goals can conflict with society norms;
- norms might conflict among themselves; and
- agents might be members of more than one society.

In light of the possibility of norm infringement and the need for autonomous agents to reason about normative societies, Lopez y Lopez *et al.* [Lopez y Lopez et al., 2004] also define reasoning mechanisms over the effects of norm compliance and violation, as well as rewards and punishments. Their model proposes methods for evaluating the benefits of joining a society as well as methods for evaluating whether to stay in a society or to leave it. An agent is seen as staying in a society for two main reasons: due to *unfulfilled goals* within the society or *social obligations*. Here, a social obligation might be that of complying with agreed norms, to reciprocate or help a fellow agent, or even coercion from another member of the society. The autonomy advocated by this model also includes mechanisms for an agent to voluntarily adopt norms; that is, an agent recognises itself as an addressee and starts following the appropriate norms. This mechanism is important, for instance in situations in which societal laws change dynamically. Finally, the model defines processes through which an agent complies with the norms by adopting or refraining from adopting intentions to achieve normative goals.

2.6 Discussion

Agent systems research has gone through a number of phases in its history, starting from a deliberative stance that put too little emphasis on interaction with the real world, focusing instead on logics and planning. This approach has been proven limited in many respects, some of which have been addressed by reactive architectures, which demonstrated that in many real-world scenarios it is imperative for agents to have simple mechanisms to deal with quick decisions (*e.g.* [Schut and Wooldridge, 2001; Wooldridge and Jennings, 1995]). This has been the main drive behind reactive architectures, which provide this property by relying on very simple rules of behaviour, virtually guaranteeing quick responses.

Here, the ability to adopt goals *in reaction* to changes in the environment allows an agent to be independent from others [Steels, 1994], whereas the ability to independently *choose* future-directed actions [Luck et al., 2003] allows an agent to deal with unforeseen situations and adapt to these changes without supervision. The behaviour of a truly autonomous agent must thus not be limited to immediate responses to a stream of events in order to start its future directed behaviour. For instance, it is not straightforward to model a Mars rover agent that has to cover multiple waypoints supplied by mission control using a simple reactive approach. Since the rover cannot go to multiple waypoints simultaneously, it must somehow *queue* the events notifying it of the waypoints in order to navigate them sequentially. This behaviour is not easily described in terms of immediate reactions. Moreover, when reacting immediately to every relevant event in the environment, there is no *choice* from alternate goals or courses of action, since the rover is carrying out an imperative *script* defined at design time.

However, we have seen that these architectures fail when an agent needs to accomplish more elaborate tasks, especially those requiring long term planning. They are thus limited, and effective architectures must be able to perform a mixture of reactive and deliberative behaviour. This problem led to the creation of a number of *hybrid* architectures that try to integrate reactive reasoning with longer term plans; in addition to the ability to adopt goals in a *reactive* way, *proactive* goal adoption is required by *autonomous* agents to pursue long-term goals [Norman and Long, 1994] using an agent's ability to anticipate the environment.

The BDI model stands in an interesting position regarding these architectures, as its philosophical underpinning is clearly deliberative, but its corresponding implementations are based on reactively activated rules. A number of autonomous agent architectures have been proposed based on the BDI model of reasoning claiming that this is enough to endow agents with autonomy, due to the fact that it provides a mentalistic mechanism for the selection of goals and commitment to their achievement. Although the BDI model is widely accepted as descriptive of the way in which humans make decisions about their actions, the agent architectures that have arisen using this model are also limited. In particular, agents created with these architectures are only able to select goals from a predefined set as a reaction to events in the environment, sometimes using additional constraints to filter irrelevant goals. This type of reactive goal adoption [Norman and Long, 1994] results in inflexible behaviour, dictated only by triggering conditions. As a result, such simple precondition-based goal selection not only represents an oversimplification of autonomous behaviour but, according to Luck *et al.* [Luck et al., 2003], is also misleading about how to achieve autonomy in agent architectures, since there is no architectural support for reasoning at runtime about the goals being pursued.

A commonly quoted statement regarding agent behaviour in a computer system is that "objects do it for free; agents do it for money" [Jennings et al., 1998]. However, this statement is misleading since the trigger-response description used almost without exception in agent languages for goal adoption entails that agents will do "for free" whatever actions they are scripted to do immediately after perceiving certain events, little different to method invocation in object programming. In contrast, we believe that agents must be able to generate their own plans at runtime or order to adapt to new situations not foreseen at design time, thus achieving this elusive ideal of autonomy in the sense of an agent freely choosing its actions. We do not, however, disregard the importance of predefined behaviours that allow an agent to quickly deal with frequent situations, but an agent architecture must also be equipped to deal with infrequent and unforeseen ones.

Now, these increasingly complex architectures, both hybrid and BDI, demand further consideration. For example, as hybrid architectures such as InteRRaP and TouringMachines (in Section 2.2.5) have very explicitly shown, integrating many types of behaviour requires some sort of internal control mechanism, able to prioritise certain types of behaviour to ensure fundamental tasks are performed first. Similarly, the vast majority of recent BDI architectures are not capable of planning, and in order for agents to generate new behaviours not seen at runtime, we need to add planning capabilities in addition to the reactive plan adoption capability that is typically already present. Moreover, we must provide agents with the means to explicitly reason about their goals so that an agent can rationally decide between investing processing time in planning. We therefore include a requirement in an architecture for such an internal control mechanism (also known as meta-level reasoning). To this end, many approaches to meta-level control have been proposed, which we surveyed in Section 2.4, and we argue that motivations provide an appropriate abstraction for meta-reasoning.

Finally, we have to consider how an agent architecture fits within a larger multiagent system, and most agent languages are focused on either the *micro level* (single agent action), or *macro level* (multiagent interaction). Languages like AGENT0 (in Section 2.3.1), for example, focus on systems that are composed of extremely simple agents whose cooperation is intended to deliver more complex functionality, whereas BDI-type architectures focus on single agent plan execution. Moreover, when designing multiagent systems, such architectures provide at most a communication language with no other support or guideline for creating dynamically collaborating agents. Even among these architectures, cooperation between agents largely assumes that a designer knows all the agents in a system at design-time, which limits the potential flexibility of open and dynamic agent systems.

An important requirement for flexible agents in a dynamic society, then, is that this flexibility also spans multiagent interaction. One way of affording this flexibility is to use multiagent planning such as PGP and GPGP (reviewed in Section 2.5.2), but these algorithms rely on a particular type of agent architecture, as well as joint goals that are to be satisfied through planning. PGP and GPGP also do not take advantage of many new planning techniques developed in recent years. Moreover, if agents are to be flexible in their dealings with other agents, one must consider that flexible agents are also assumed to be self-interested, thus requiring some form of societal control. We have reviewed norms (in Section 2.5.3) as a possible mechanism to achieve this.

The architectures we have seen, therefore, do not have the ability to take advantage of agents containing capabilities not foreseen at design time by creating new plans including them. Existing efforts on distributed planning algorithms usually assume shared knowledge of all possible tasks and actions for the agents involved. What is needed, therefore, is a mechanism that allows agents to discover new capabilities and compose new plans at runtime that use them, thus increasing the flexibility of the system as a whole. Furthermore, the increased flexibility afforded by these new plans means that it is impossible to foresee all potential new behaviours resulting from the composition of multiple agent's capabilities. This in turn leads to a requirement for some societal control mechanism that ensures undesirable behaviours are either not possible, or if possible, that agents engaging in them are penalised to ensure an overall positive outcome for the system.

Chapter 3

Plan Generation in AgentSpeak(PL)

3.1 Introduction

As we have seen in the discussion of Section 2.6, in order to adapt to situations not initially foreseen by the designer, agents need the ability to generate new plans at runtime, in addition to a set of predefined plans to react to common situations. The ability to generate new plans, however, requires a re-examination of the way in which goals are structured in BDI agent architectures. There are two ways of considering goals:

- as desired states of affairs; or
- as procedures to be executed.

Conceptually, the *desires* component of the BDI model refers to a desired *state of affairs*, regardless of the way in which this state of affairs is achieved, whereas the *intentions* correspond to courses of action selected to *achieve* selected desires. When moving from the conceptual level to practical agent architectures, this type of desire or goal is generally denominated a goal *to be* [Winikoff et al., 2002] and is declarative in nature, since there is no automatic commitment to a specific plan to bring it about. By contrast, in the latter case, when an agent commits to *achieving* a goal, it commits to some plan that will bring about this goal to be; this plan commitment is generally called a goal *to do* [Winikoff et al., 2002], because it implies commitment to a *procedure*. As we have seen in Chapter 2, most agent architectures consider only this latter *procedural* approach to goals.

Such a *procedural* response to goal achievement has generally been favoured to enable the construction of *practical* systems that are usable in real-world applications. However, it also makes agents inflexible in cases of failure. When a procedural agent selects a plan to achieve a given goal it is possible that the selected plan may fail, in which case the agent typically concludes that the goal has also failed, regardless of whether other plans to achieve the same goal might have been successful. By neglecting the *declarative* aspect of goals in not considering the construction of plans on-the-fly, agents lose the ability to reason about alternative means of achieving a goal, making it possible for poor plan selection to lead to an otherwise avoidable failure.

Typically, agents select plans using more or less elaborate algorithms, but these seldom have any knowledge of the content of the plans, so that plan selection is ultimately achieved using fixed rules, with an agent adopting *black box* plans based solely on the contextual information that accompanies them. For example, in PRS and AgentSpeak(L) [Georgeff and Lansky, 1987; Rao, 1996], plans consist of procedures with a header indicating when these procedures are to be executed. Alternatively, rather than selecting from existing plans, some agent interpreters use plan modification rules to allow plans to be modified to suit the current situation [van Riemsdijk et al., 2003], but this approach still relies on a designer establishing a set of rules that considers all potentially necessary modifications for the agent to achieve its goals. The problem here is that for some domains, an agent description must either be exhaustive (requiring a designer to foresee every possible situation the agent might find itself in), or will leave the agent unable to respond under certain conditions.

Our aim in this chapter is to address this problem and create an agent architecture that can generate new plans at runtime to deal with new situations not foreseen at design time. We call this architecture AgentSpeak(PL), for *planning* AgentSpeak(L). Besides generating new plans, this architecture must be able to perform common tasks, foreseeable at design time, quickly without the need to spend time planning to achieve each goal, as well as reusing previously generated plans. In this architecture, plans are built by chaining existing fine-grained plans from a plan library into high-level plans. We ensure the practicality of the architecture by building it on top of an existing procedural agent architecture, and the resulting architecture provides for a combination of declarative and procedural aspects. Our approach relies on an underlying planning component, which requires us to examine how the agent architecture and the planning component interoperate, resulting in a new agent architecture, but with minimal modification of the original one. Moreover, this modification requires no change to the plan language, allowing designers to specify predefined procedures for known tasks under ideal circumstances, but also allowing an agent to form new plans when unforeseen situations arise. The key contribution of the system lies in its runtime flexibility, allowing an agent to use its plan library to respond to new situations without the need for the designer to specify all possible combinations of low-level operators in advance.

We start the chapter by introducing the building blocks of our approach through a description of the planning formalism used throughout, setting up the theoretical basis for planning, and the planning language used in the connection between agent and planner in Section 3.2. We then proceed to describing the AgentSpeak(L) architecture in Section 3.3, and the notation of its associated programming language. After these concepts are presented, we proceed to explain the rationale of our approach by comparing AgentSpeak(L) plans to planning operators in Section 3.4, followed by the two key processes in our technique that connect an agent to a planner and *vice versa* in Sections 3.5 and 3.6. The validation of our approach is achieved through experiments described in Section 3.7 that compare planning agents to traditional ones, as well as a comparison with related work in Section 3.8. Finally, we draw some conclusions about our planning architecture, its limitations and possible improvements in Section 3.9.

3.2 AI Planning

When needing to accomplish complex tasks that involve more than one step, humans make *plans* to determine which steps are needed to accomplish a task and which step to take first, as well as the order of subsequent steps. Although it is relatively easy to select actions for immediate execution, or to execute predetermined plans in response to events, generating new plans is not a simple process. Indeed, early AI research was concerned with this problem of how to concatenate individual actions to achieve a particular objective, in a process also known as *means-ends* reasoning [Ghallab et al., 2004], or planning from first principles. This resulted in a number of approaches that eventually led to the development of generic planning systems (or planners) [Fikes and Nilsson, 1971].

Since our goal is to add plan generation capabilities to an agent architecture, it is necessary to understand the process through which plans are created, and we therefore review AI planning in this section. We start with a discussion of the background of planning techniques and follow with a description of the formalism used to describe planning problems and a description of the planning process, finishing with a planning example.

Initial approaches to general purpose planning include the *Stanford Research Institute Problem Solver* (STRIPS) [Fikes and Nilsson, 1971], whose notational concepts are still used as the basis for the specification of planning problems, as well as multiple approaches to *Partial Order Planning* (POP) [Ambros-Ingerson and Steel, 1988]. These approaches to generic planning were very limited in the size and type of problems that could be handled in a reasonable amount of time, due to their method of navigating the search-space. After a lull in new approaches to planning, several new algorithms were developed, such as Graphplan [Blum and Furst, 1997], SATPlan [Kautz and Selman, 1992] and HTN [Erol et al., 1994], representing a significant leap of efficiency, and allowing more complex planning problems to be computed in reasonable time [Weld, 1999].

3.2.1 Planning Problem Specification

Most planning formalisms are derived from the problem description language used by STRIPS [Fikes and Nilsson, 1971]. In general, a planner takes three inputs, with these three inputs constituting a *planning problem specification*: a description of the initial state of the world; a description of the goal state that should be true after a plan is executed; and a description of the domain in terms of the set of available operators (actions), in some formal language (such as the language of STRIPS). States are generally represented as *logic atoms* denoting what is true in a certain world, and the planner tries to generate a sequence of actions which, when applied to the initial state, modifies the world so that the goal state becomes true. This process is illustrated in Figure 3.1, which shows how a *problem description*, consisting of an initial and a goal state, and a *domain description*, consisting of the operators, are supplied to a planning process to generate a plan.

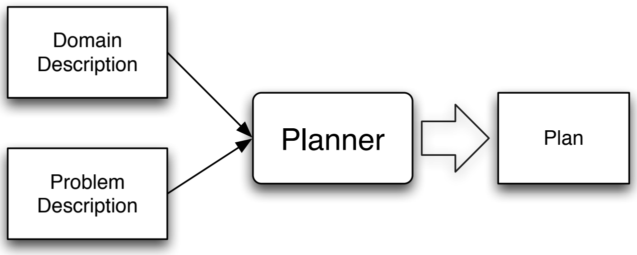


FIGURE 3.1: Summary of the planning process.

A planning algorithm solves a problem by finding a sequence of instantiated operators that transform the world from an initial state to a goal state. The specification is used to generate the search-space over which the planning system searches for a solution, where this search-space consists of all possible instantiations of the set of operators using the *Herbrand universe*[1] derived from the symbols contained in the initial and goal state specifications.

This sequencing process is usually based on selecting operators whose preconditions are valid in the initial state or were made valid by the effects of previous operators until a partially ordered set of operators is selected, such that their application transforms the initial state into the goal state. In STRIPS, the effect of an operator is described in terms of *add lists* denoting the atoms that become true as a result of the operator being executed, and *delete lists* denoting the atoms that cease to be true as a result of the operator being executed. For example, if an operator has an add list that contains the atoms a and b, and a delete list that contains the atom c, the execution of this operator will result in a and b being true and c being false in the following world-state. An illustration of how operators are sequenced and ordered is shown in Figure 3.2.

The original version of the STRIPS language was later extended for more expressivity in various ways, with no preoccupation of formally relating all the extensions, resulting in languages like the Action Description Language (ADL) [Pednault, 1989] and the Planning

[1]Any formal language with symbols for constants and functions has a Herbrand universe, which describes all of the terms that can be created by the application of all combinations of constant symbols as parameters to all functional symbols.

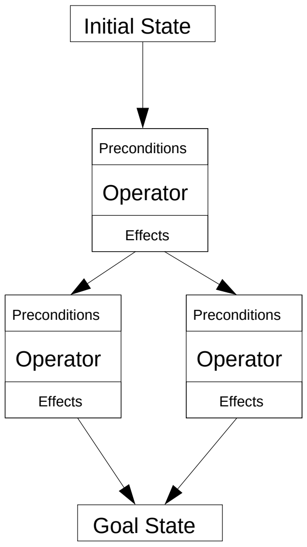


FIGURE 3.2: Overview of a plan as a sequence of partially ordered operators.

Domain Description Language (PDDL) [Fox and Long, 2003][2]. Nebel, however, defined a formal framework in which all STRIPS-related planning specifications could be related [Nebel, 2000], and we adapt this formalism in our work [Meneguzzi et al., 2007]. We describe this formalism in the following sections, starting with the logic language that underpins the formalism in Section 3.2.1.1 followed by the formalism itself, with the states in Section 3.2.1.2, and operators and plans in Section 3.2.1.3.

3.2.1.1 The logic language (Literals, Terms, Negation)

Let us consider a first-order logic language consisting of an infinite set of symbols for predicates, constants, functions and variables, obeying the usual formation rules of first-order logic. For simplicity of presentation, we refer to well-formed atomic formulas as *atoms*. Let $\mathbf{\Sigma}$ be the infinite set of atoms and variables in this language and let Σ be any finite subset of $\mathbf{\Sigma}$. From these sets, we define $\hat{\Sigma}$ to be the set of *literals* over $\mathbf{\Sigma}$, consisting of *atoms* and *negated atoms*, as well as constants for truth ($\top$) and falsehood ($\bot$). We denote the *logic language* over the logical connectives of conjunction ($\wedge$) and negation ($\neg$) as $\mathcal{L}_\Sigma$. Furthermore, considering a set L of literals, we denote the positive literals in this set as $pos(L)$, meaning that all literals in this set are explicitly true, and the negative literals as $neg(L)$, meaning that all literals in this set are explicitly false. We summarise the notation used in this section in Table 3.1.

[2]The original version of PDDL was created by McDermott, but not published in any public forum.

Notation	Meaning
$\bot$	falsehood
$\top$	truth
$pos(L)$	positive literals in L
$neg(L)$	negative literals in L
$pre(o)$	preconditions of operator o
$post(o)$	post-conditions of operator o
$A \models B$	A entails B

TABLE 3.1: Summary of notation for Section 3.2.1.

3.2.1.2 States

In order to describe the world, we use the logic language to represent what is true and what is not true at any given time. A *state* s is a truth-assignment for atoms in Σ, and a *state specification* S is a subset of $\hat{\Sigma}$ specifying a logic theory consisting solely of literals. S is said to be complete if, for every literal l in Σ, either l or $\neg l$ is contained in S. A state specification S describes all of the states s such that S logically supports s. For example, if we consider a language with three atoms a, b, and c, and a state specification $S = \{a, \neg b\}$, this specification describes the states $s_1 = \{a, \neg b, c\}$, and $s_2 = \{a, \neg b, \neg c\}$. In other words, a state specification supports all states that are a model for it, so a complete state specification has only one model.

The specification formalism we use allows incomplete state specifications and first-order literals on the preconditions and effects of planning operators (incomplete state specifications can omit predicates that are not changed by an operator from its preconditions and effects, as opposed to requiring operators to include every single predicate in the language's Herbrand base[3]).

3.2.1.3 Operators and Plans

Planning is concerned with sequencing actions, and in a planning specification, actions are generated by instantiated abstract operators. Operators describe state transformations, and following Nebel [Nebel, 2000], we define them as pairs $o = \langle pre, post \rangle$, for each operator o, where $pre(o) \in \mathcal{L}_\Sigma$ and $post(o) \in \mathcal{L}_\Sigma$ respectively denote pre-conditions and post-conditions of o. The result of applying an operator o with post-conditions $post(o)$ to a consistent state specification S is a new state specification S' in which the post-conditions $pos(post(o))$ are true. In terms of set operations, S' is the result of adding the literals of $pos(post(o))$ to S, while removing $neg(post(o))$, as formally stated in Definition 3.1. This is roughly equivalent to *add* and *delete* lists from STRIPS, with the main difference being that, in our formalism,

[3]Any formal language with a Herbrand universe and predicate symbols has a Herbrand base, which describes all of the terms that can be created by applying predicate symbols to the elements of the Herbrand universe.

it is possible to explicitly assert the falsehood of a given term, *e.g.* regardless of the existence of a literal a, we may have an operator that asserts that a becomes explicitly false after its execution, resulting in $\neg a$.

Definition 3.1 (Function R)**.** The result of applying an operator o to a state specification S is defined by:[4]

$$R : 2^{\hat{\Sigma}} \times \mathbf{O} \to 2^{\hat{\Sigma}}$$

$$R(S,o) = \begin{cases} S \setminus \neg neg(post(o)) \cup post(o), & \text{if } S \models pre(o) \\ & \text{and } S \not\models \bot \\ & \text{and } post(o) \not\models \bot; \\ \bot, & \text{otherwise.} \end{cases}$$

Using function R, it is possible to specify the result of applying finite sequences of operators $\mathbf{O}^*$ to an initial state specification, in which elements Δ of $\mathbf{O}^*$ are plans, formally stated in Definition 3.2.

Definition 3.2 (Function Res)**.** The result of successively applying operators to an initial state is defined by:

$$\begin{aligned} Res : 2^{\hat{\Sigma}} \times \mathbf{O}^* &\to 2^{\hat{\Sigma}} \\ Res(S, \langle\rangle) &= S \\ Res(S, \langle o_1, o_2, \ldots, o_n \rangle) &= Res(R(S, o_1), \langle o_2, \ldots, o_n \rangle) \end{aligned}$$

As we have seen, a planning problem following the STRIPS paradigm comprises a domain specification, and a problem description stating the initial world-state and the goal world-state. By using the definitions introduced in this section, we can define a planning instance formally, as stated in Definition 3.3. Here, the solution for a planning instance is a sequence of operators Δ which, when applied to the initial state specification using the *Res* function, results in a state specification that supports the goal state. The solution for a planning instance or *plan* is formally defined in Definition 3.4.

Definition 3.3 (Planning Instance)**.** A planning instance is a tuple $\Pi = \langle \Xi, \mathbf{I}, \mathbf{G} \rangle$, in which:

- $\Xi = \langle \Sigma, \mathbf{O} \rangle$ is the domain structure, consisting of a finite set of atoms Σ and a finite set of operators $\mathbf{O}$;
- $\mathbf{I} \subseteq \hat{\Sigma}$ is the initial state specification; and

[4] We use the notation 2^S to denote the *power set* of S [Weisstein, 1999].

- $\mathbf{G} \subseteq \hat{\Sigma}$ is the goal state specification.

Definition 3.4 (Plan)**.** A sequence of operators Δ is said to be a plan for a planning instance Π, or the solution of Π, if and only if $Res(\mathbf{I}, \Delta) \not\models \perp$ and $Res(\mathbf{I}, \Delta) \models \mathbf{G}$.

A planning function is a function that takes a planning instance as its input and returns a plan for this planning instance, or else returns an empty set indicating that no plan exists for this instance. This is stated formally in Definition 3.5.

Definition 3.5 (Planning Function)**.** A planning function that takes a planning instance Π as input is defined as:

$$Plan : \Pi \rightarrow \mathbf{O}^*$$

$$Plan(\Pi) = \begin{cases} \Delta, & \text{if } \exists\Delta \text{ and } \Delta \text{ is a solution of } \Pi; \\ \langle\rangle, & \text{otherwise.} \end{cases}$$

Now that we have formally described the notion of a plan and a planner, we need to understand the agent formalism in order to establish a link between an agent reasoning cycle and a planning function, which we do in Section 3.3.1. First however, it is useful to see how these concepts are actually used in a real planning problem, which we do next.

3.2.2 Planning Example

To illustrate how problems are described using a STRIPS-like specification, we consider a hypothetical postman robot agent responsible for delivering packets when they arrive in a warehouse through loading bays, as illustrated in Figure 3.3. Whenever a packet arrives, the robot picks up the new packet and delivers it to the relevant pigeonhole on the opposite side of the warehouse. The robot operates on battery power, which is expended as it moves through the warehouse; whenever its charge reaches a critical level, the robot moves to a charger located in a corner of the warehouse in order to replenish its charge. Given such a postman robot, we can specify a potential set of operators that would enable this robot to plan to achieve the various objectives of this scenario.

Throughout this thesis, we use a textual representation of STRIPS problems in which operators are preceded by the keyword `operator`, preconditions by the keyword `pre` and postconditions by the keyword `post`. Operator headers may contain variables following the Prolog notation [Nilsson and Maluszynski., 1995] where constants are written with a lower-case first character, while variables are written with an upper case first character, so that $position1$ is a constant and $Position$ is a variable. All variables used in the preconditions and effects must be declared in the operator header, so that when an operator is instantiated, preconditions and effects become ground. Thus, an operator called **op**(X), with preconditions a(X) and b(X), and an effect c(X), would be represented as shown in Table 3.2.

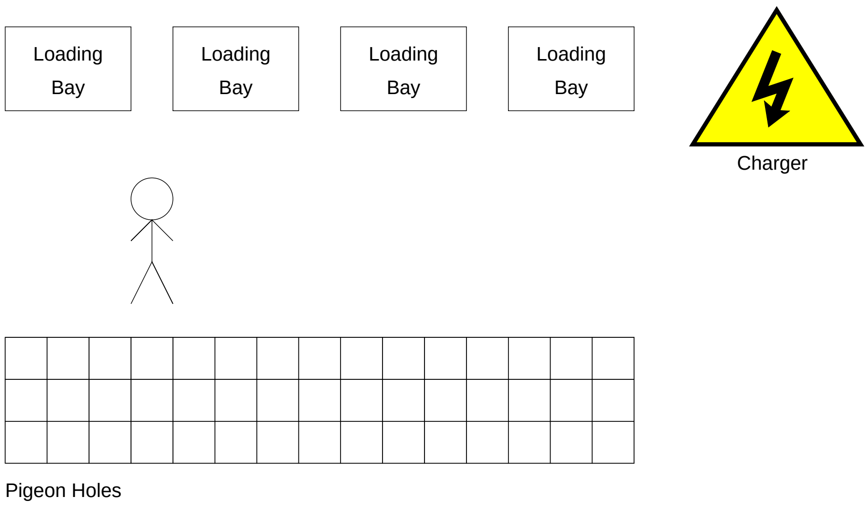


FIGURE 3.3: Postman robot scenario.

```
operator op(X)
pre: a(X) & b(X)
post: c(X)
```

TABLE 3.2: Textual representation of a STRIPS operator.

We start by defining the operator associated with movement, shown in Lines 1-3 in Table 3.3, which is defined in the usual way for STRIPS problems, specifying that the robot must be at an initial position, different from the destination and, at the end of operator execution, ceases to be at this position and reaches the destination. We add the additional constraint that the robot's battery must not be empty.

The next two operators refer to picking up and dropping packets, shown in Lines 5-11 in Table 3.3. These operators are complementary in representing the robot's hold of a certain packet, and require the agent to be positioned at the same place as the packet for the pickup and at the target location for the drop.

Finally, we represent the operator that allows the agent to charge its battery in Lines 13-15 in Table 3.3, requiring the agent to be at the charger, and resulting in the agent's battery being replenished.

It is important to note that Table 3.3 shows our example in a language commonly used by *implementations* of planners, and the correspondence of this language with the formalism introduced in Section 3.2.1 should be fairly straightforward. However, to make this correspondence explicit, we summarise the formal components of each operator in Table 3.4.

Using the domain specification of Table 3.3, we can now specify a planning problem for which a plan should be generated. We use the scenario of the arrival of a new packet at a loading bay, with the agent's goal being to deliver that packet to the pigeonholes. This

```
operator move(A,B)
pre: at(A) & not at(B) & not batt(empty)
post: at(B) & not at(A)

operator pickup(Packet, Position)
pre: at(Position) & over(Packet,Position) & not held(_)
post: held(Packet) & not over(Packet,Position)

operator drop(Packet, Position)
pre: at(Position) & held(Packet)
post: not held(Packet) & over(Packet,Position)

operator charge
pre: at(charger)
post: batt(full)
```

Table 3.3: Postman robot STRIPS specification.

Operator	**Preconditions**	**Post-Conditions**
move(A,B)	at(A) not at(B) not batt(empty)	at(B) not at(A)

pickup(Packet, Position)	at(Position)
	over(Packet,Position)
	not held(_)
		held(Packet)
		not over(Packet,Position)

drop(Packet, Position)	at(Position)
	held(Packet)
		not held(Packet)
		over(Packet,Position)

charge	at(charger)	batt(full)

Table 3.4: Postman robot STRIPS specification.

situation is expressed in the initial state (start) of Line 1 and the goal state of Line 2 in Table 3.5.

```
1 start: at(position1) & batt(full) & over(packet1, bay1)
2 goal: over(packet1, pigeonHoles)
```

Table 3.5: Example problem.

Given this planning problem specification, one possible plan involves: moving the agent from its initial position at bay1; picking up packet1; moving to the pigeonholes; and dropping the packet, as shown in Table 3.6.

```
move(position1, bay1).
pickup(packet1,bay1).
move(bay1,pigeonHoles).
drop(packet1,pigeonHoles).
```

Table 3.6: Plan for the problem of Table 3.5.

3.3 AgentSpeak

Now that we have specified planning, we need to examine the agent architecture, language, and interpreter used as the basis for our work. As outlined in Chapter 2, AgentSpeak(L) [Rao, 1996] is an agent language that allows a designer to specify a set of procedural plans which are then selected by an interpreter to achieve the agent's design goals. It evolved from a series of procedural agent architectures originally developed by Rao and Georgeff [Rao and Georgeff, 1995a]. In AgentSpeak(L), an agent is defined by a set of beliefs and a set of plans, with each plan encoding first a procedure that is assumed to bring about a desired state of affairs, and second a header that says when the plan is to be adopted. Goals in AgentSpeak(L) are implicit, and plans intended to fulfil them are invoked whenever some triggering condition is met in a certain context, notionally the moment at which this implicit goal becomes relevant.

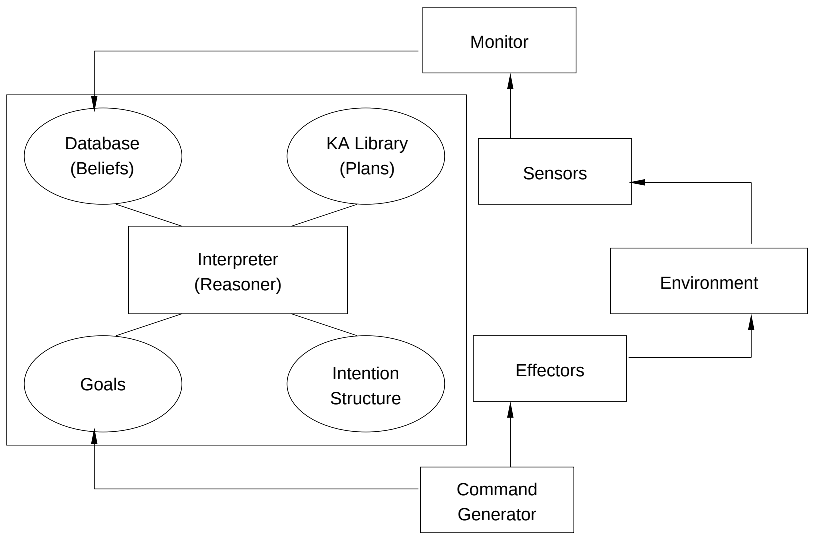


FIGURE 3.4: Overview of the PRS architecture.

Concrete AgentSpeak(L) agents follow the same architecture as the Procedural Reasoning System (PRS) [Georgeff and Ingrand, 1989a], from which its operational semantics is derived. The architecture is illustrated in Figure 3.4, which shows the main data components of a PRS agent, where circles represent data structures and rectangles represent processes. The four data structures are:

- *beliefs*, comprising the information known by the agent, and regularly updated as a result of agent perception;
- *plans*, representing the behaviours available to the agent, as well as the situations in which these plans are applicable;
- *goals*, representing situations to which the agent will react by adopting plans, corresponding to desired world-states; and

- *intention structures*, comprising the set of partially instantiated plans currently adopted by the agent.

These data structures are used by the interpreter during its control cycle, as described in Section 3.3.2.

3.3.1 Language

AgentSpeak(L) agents (or *agent programs* [Rao, 1996]) are specified as a set of initial *beliefs* and a set of *plans* that comprise the agent's plan library. The AgentSpeak(L) language uses first-order logic predicates (as well as constants for truth and falsehood) as building blocks for the specification of belief formulae, plan headers, and action invocations. If p is a predicate symbol and $\mathbf{t}$ is a (possibly empty) set of terms $t_1, \ldots, t_n$, then $p(t_1, \ldots, t_n)$ is an AgentSpeak(L) predicate. Terms can be either constant (representing elements in the world) or variables to be bound during agent execution, and follow the Prolog convention [Nilsson and Maluszynski., 1995]. If a predicate contains only constants, the predicate is said to be *ground*.

3.3.1.1 Beliefs

Individual predicates representing beliefs are called *belief atoms*, and can be combined to form *beliefs* using the logic connectives of negation (*not*) and conjunction ($\wedge$ or &). For example, at(position1) is a belief atom, and **not** at(position1) & at(position2) is a belief.

3.3.1.2 Goals and Events

There are two types of AgentSpeak(L) goal: *achievement* goals, represented by a predicate preceded by an exclamation mark (*e.g.* !move(A,B)); and *test* goals, represented by a predicate preceded by a question mark (*e.g.* ?at(Position)). Test goals are used by an agent to verify whether a given belief is true, whereas achievement goals are used by an agent to achieve a certain state of affairs. Though Rao [Rao, 1996] describes AgentSpeak(L) goals as representing world-states that an agent wants to achieve, as we have discussed above they can more precisely be described as intention headers used to identify groups of plans intended to achieve an implicit objective.

An agent is notified of changes perceived in the environment as well as modifications to its own data structures through *triggering events*, which may trigger the adoption of plans by an agent. There are four types of events, consisting of the addition and deletion of beliefs and goals. Addition is denoted by the plus (+) sign, while deletion is denoted by the minus ($-$) sign so, for example, +!move(home,office) denotes the addition of a goal to move from home to the office, and $-$at(home) denotes the deletion of the belief that an agent is at home.

3.3.1.3 Actions and Plans

An agent interacts with the environment through *atomic actions*, which are executed in the environment in order to achieve some effect. Actions are invoked by an agent during the course of executing a plan using predicates comprised of an action name and its parameters. Thus if walk is an action to walk from one place to another and takes two parameters, an AgentSpeak(L) invocation of this action to walk from home to the office would be denoted by walk(home, office).

Finally, an agent's behaviours are expressed through plans which specify the means for achieving particular (implicit) goals whenever certain events occur under certain circumstances. Plans comprise a *head* describing the conditions under which the plan should be adopted, and a *body* describing the actions that an agent should carry out to accomplish the plan's goal. A plan head contains two elements, an invocation condition, which describes when the plan becomes *relevant* as a result of a triggering event; and the *context* in which the plan is *applicable*, specified as a formula over an agent's beliefs. A plan is specified as follows: if e is a triggering event (that should match a certain invocation condition), $b_1 \wedge \cdots \wedge b_n$ are belief literals, and $h_1, \ldots, h_n$ are goals or actions, then '$e : b_1 \wedge \cdots \wedge b_n \leftarrow h_1; \ldots; h_n$.'[5] is a *plan*. If a plan is to be invoked whenever its invocation condition occurs, the context part of the plan contains only the *true* literal. The structure of an AgentSpeak(L) plan specification is illustrated in Figure 3.5.

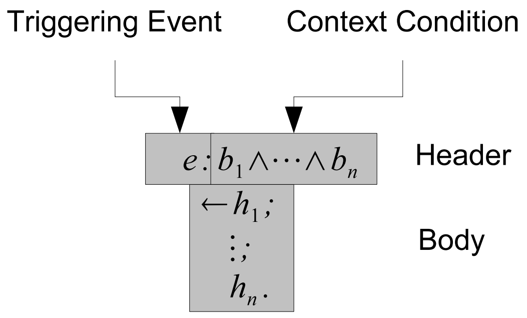


FIGURE 3.5: Components of an AgentSpeak(L) plan specification.

3.3.1.4 Intentions

Plans that are instantiated and adopted by an agent are called *intentions*, and are stored by an agent in its *intention structure*. Here, when an agent adopts a certain plan, it is committing itself to executing the plan to completion. For example, Table 3.7 illustrates an AgentSpeak(L)[6] plan to walk between two locations (From and To) under certain conditions.

[5]We use quotation marks here to emphasise that a plan finishes with a full stop.

[6]In this chapter, we use AgentSpeak and AgentSpeak(L) in distinct ways: when explicitly referring the *programming language* we always use AgentSpeak(L), whereas AgentSpeak refers to any AgentSpeak interpreter.

The plan is considered by the agent whenever it generates an event to adopt a goal !move(From,To), and carried out if the agent believes it is not already at its destination and does not have a car. The plan itself consists of one subgoal to find its shoes before carrying out an action to walk between the desired locations. As illustrated in Figure 3.6, adopting a subgoal entails the execution of another plan (and in turn the addition of this new plan to the same intention as the one containing the original plan to move), in this case a plan with an invocation condition of +!find(shoes), while the action to walk is executed directly by the agent.

```
+!move(From,To) : at(From) & not at(To) & not has(car)
    <- !find(shoes);
         walk(From,To).
```

TABLE 3.7: Example of an AgentSpeak(L) plan.

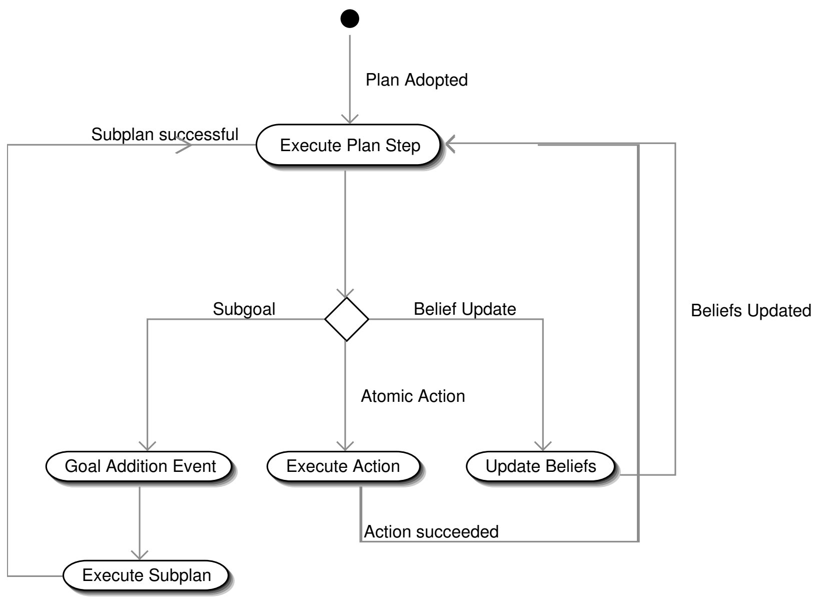


FIGURE 3.6: Executing AgentSpeak plan steps.

It is important to note that the intention structure may contain multiple intentions, organised in a hierarchical way. Each individual intention is a stack of steps that must be executed, with the next executable step at the top of the stack. Intentions adopted as a reaction to events in the environment are said to be at the top of the intention structure, so their steps can be immediately executed. Intentions adopted to achieve subgoals of an existing intention are stacked on top of the subgoal, so that their steps are executed before the rest of the original intention.

3.3.2 Interpreter and Control Cycle

At every control cycle, a list of events is generated by the interpreter, which may originate from changes in the environment or the execution of intentions. Changes in the environment are handled by the *belief revision function* (shown as the BRF component in Figure 3.7), which is responsible for modifying the belief database to reflect the state of the world. The events generated by the environment are passed to the agent, which invokes the *event selection function* (shown as the S_E component in Figure 3.7) to select one of these events. Once an event has been selected by S_E, AgentSpeak tries to match the event with the invocation condition of plans from the plan library (shown as the *Unify Event* process in Figure 3.7) yielding a set of *relevant plans.* AgentSpeak then attempts to match the context condition of these plans with the current beliefs (shown as the *Unify Context* process in Figure 3.7) to determine a set of *applicable plans*, which are sent to the *plan selection function* (shown as the S_O component in Figure 3.7) to select one of these plans to be added to the agent's intention structure. New plans are added to the top of the intention structure if they were adopted as a result of an *external event* (arising from changes in the environment), or to the intention that generated the *internal event* that triggered this plan's adoption.

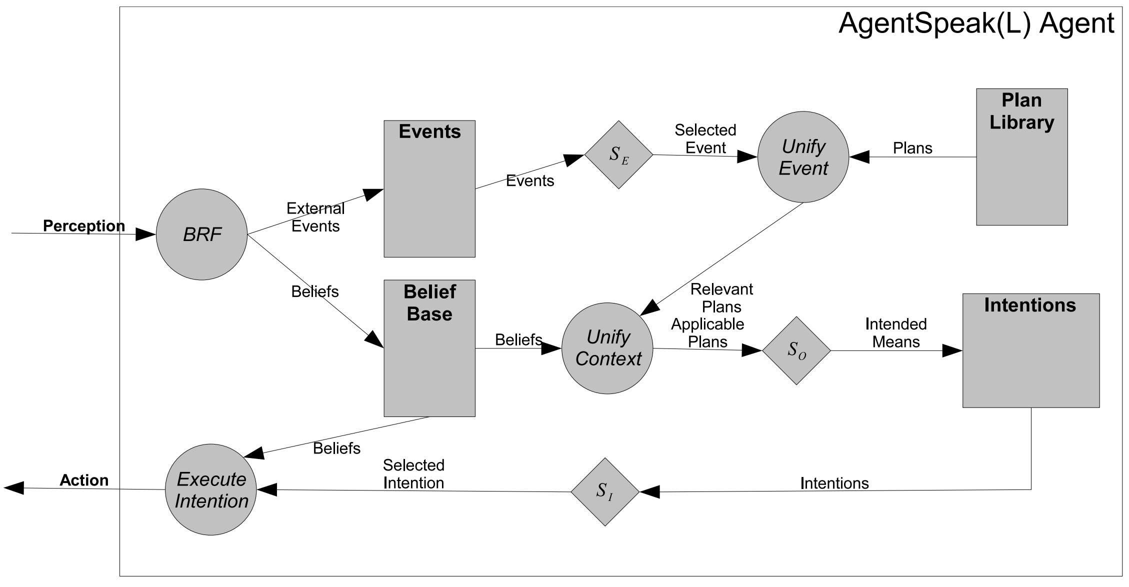


Figure 3.7: AgentSpeak control cycle [Bordini et al., 2005a].

As there can be multiple external events happening simultaneously in the environment, multiple parallel intentions can be created at the top of the intention structure. Thus, at the end of the control cycle, an agent inspects its intention structure and selects an intention using the *intention selection function* (shown as the S_I component in Figure 3.7) for execution. Since each intention consists of a stack of partially instantiated plans, the execution of an intention consists of taking the next step of the topmost plan in the stack and either executing it (if the step is an atomic action), generating an internal event (if the step is an achievement goal), or querying the belief base (if the step is a test goal). Once a plan step is executed, it is removed from the intention. When an intention has no further steps remaining to be executed, it is removed from the intention structure and is considered to have been accomplished by the agent. If any step of a plan fails to be executed, such as an action failing to execute, a test goal not unifying with any predicate in the belief base, or a subgoal failing, the intention containing that step is considered to have failed. Thus, if a subgoal at the end of a chain of other higher-level goals fails, this failure cascades all the way through to the initial plan, invalidating all the plans adopted in the chain. Figure 3.7 illustrates this control cycle in detail, with rectangles representing sets or databases, circles representing processes, and diamonds representing selection functions.

Now, as we have seen, multiple events may occur simultaneously in the environment, and multiple intentions may be created by an agent as a result of these events leading to multiple plans becoming applicable and being adopted by the agent. So two execution outcomes are possible: interleaved or atomic execution. In the former, plans in different intentions alternate the execution of their steps, in which case care must be taken to ensure that any two plans that may execute simultaneously do not have steps that jeopardise the execution of one another. Conversely, if plans are executed atomically, it is not possible for one plan to interfere with another. AgentSpeak(L) provides no explicit mechanism to deal either with multiple plans or with the possible interference among them that may arise as a result of interleaving their execution. However, specific AgentSpeak(L) interpreter implementations may try to address this problem, and in this thesis we adopt the convention of the Jason interpreter [Bordini et al., 2007], which provides a plan directive that instructs the interpreter to execute any given plan in its entirety before considering other intentions. This allows critical plans to be executed atomically without interference from other plans, thus avoiding plan interference. Throughout the rest of this thesis, we use the Jason interpreter, and provide details as needed, at the relevant time.

3.3.3 Postman Scenario

Returning to our postman example (from Section 3.2.2), a possible AgentSpeak(L) specification for it is shown in Table 3.8, in which the robot has plans to move to the relevant loading bay, pick up the packets and move them to the pigeonholes where they are delivered (Lines 10 through 17 in Table 3.8). Moreover, the robot has a limited amount of battery power, which is expended whenever the robot moves (Lines 1 through 8 in Table 3.8), and must be recharged when it reaches a critical level (Lines 19 through 24 in Table 3.8).

So, for example, if an agent detects that a new packet (packet1) has been deposited in a loading bay (bay1), represented by the event +over(packet1, bay1), the agent immediately adopts a plan that contains the goal of delivering this packet (the achievement goal in Line

8 of Table 3.8). Since the plan has an *external event* as its invocation condition (*i.e.* +over (packet1, bay1)), when the plan is added to the agent's intentions, it becomes the root of a new intention, whereas the goal (and plan) to deliver the package, when adopted, is pushed on top of the former intention structure. The agent accomplishes this goal by executing the plan (the only plan for this goal, in Line 10 of Table 3.8), which consists of moving from its current location to the loading bay (through another goal that deals with agent movement), picking up the packet (through the atomic action pickup(Packet)), and moving it to the pigeonhole where the packet is dropped (through another atomic action, drop(Packet)).

```
 1  +!move(A,B) : A = B %Plan 1
 2      <- true.
 3  %Plan 2
 4  +!move(A,B) : at(B)
 5      <- true.
 6  %Plan 3
 7  +!move(A,B) : at(A) & not at(B) & not batt(empty)
 8      <- move(A,B).
 9  %Plan 4
10  +over(Packet,Bay) : true
11      <- !deliver(Packet).
12  %Plan 5
13  +!deliver(Packet) : packet(Packet) & over(Packet,Bay) & at(A)
14      <- !move(A,Bay);
15         pickup(Packet);
16         !move(Bay,pigeonHoles);
17         drop(Packet).
18  %Plan 6
19  +batt(critical) : true
20      <- !charge.
21  %Plan 7
22  +!charge : at(A)
23      <- move(A,charger);
24         charge.
```

Table 3.8: Postman robot specification.

As mentioned earlier, the robot can only move to perform its delivery activities when its battery charge is not empty (expressed in the context condition of the plan to achieve the !move goal, in Line 7 of Table 3.8). If the agent reaches a condition in which its battery charge becomes critical, the agent adopts the goal of charging it (through the plan in Line 19 of Table 3.8), which causes the agent to move to the charging station (charger), and invoke the atomic action charge.

Although our postman specification now seems fine, closer inspection reveals that it contains a significant limitation that manifests itself whenever the battery becomes critical during a delivery operation. This will cause the agent to adopt the plan to charge its battery, thus interfering with any existing intention that requires the agent to be in a certain position. In order to solve this problem, the plans must be carefully tailored to cater for interference from the remainder of the plan library. We illustrate this problem in Table 3.9, which depicts the stream of events from the environment in the left-hand side against the resulting response

	Environment Stream	**Agent Stream**
1	+over(packet1,bay1)	+!deliver(packet1)
2		+!move(position1,bay1)
3		move(position1,bay1)
4		pickup(packet1)
5	+batt(critical)	+!move(bay1,pigeonHoles)
6		+!charge
7		move(bay1,charger)
8		charge
9	*%Agent is no longer in bay1*	move(bay1,pigeonHoles)

Table 3.9: Plan interference trace.

from AgentSpeak(L). Here, the arrival of packet1 in bay1 triggers the adoption of the plan to deliver it to the pigeonHoles. However, during the execution of the steps from this plan, the robot's battery reaches a critical level, triggering the plan to charge its batteries, and jeopardising the assumption of a fixed position from the plan to deliver the packet.

3.3.4 Summary

As we have seen in Section 3.3, the behaviour of traditional AgentSpeak(L) is based on inflexible rules. If we wish to create complex plans to be carried out in a dynamic world, then we must foresee every possible interaction for these plans, or risk harmful interference, since no modification or analysis of the plan library is expected at runtime. This limitation can be seen, for instance, in the example of Table 3.8, where the execution of a plan to move a postman robot to a certain position is interrupted by a plan to address the near depletion of its battery charge. However, by expanding AgentSpeak(L) with planning capabilities, we allow an agent to refine its plan library at runtime as needed.

As described previously, an AgentSpeak(L) interpreter is driven by events on the agent's data structures, in the form of belief or goal additions or deletions. These events function as triggers for the adoption of plans in certain contexts, causing plans to be added to the intention structure from which the agent handles the plan's steps. Plan steps may include the execution of an atomic action, causing some change in the environment, or the addition of new goals, causing new plans to be added to the intention structure. In both situations, the agent might *fail*, either to execute an atomic action or to accomplish a subgoal, resulting in the failure of the original plan in the intention structure. If a plan selected for the achievement of a given goal fails, the default behaviour of an AgentSpeak(L) agent is to conclude that the goal that caused the plan to be adopted is not achievable.

This control cycle (summarised in Figure 3.8) strongly couples plan execution to goal achievement. It also allows for situations in which the poor selection of a plan leads to the failure of a goal that would otherwise be achievable through a different plan in the plan library. While such limitations can be mitigated through meta-level constructs that allow

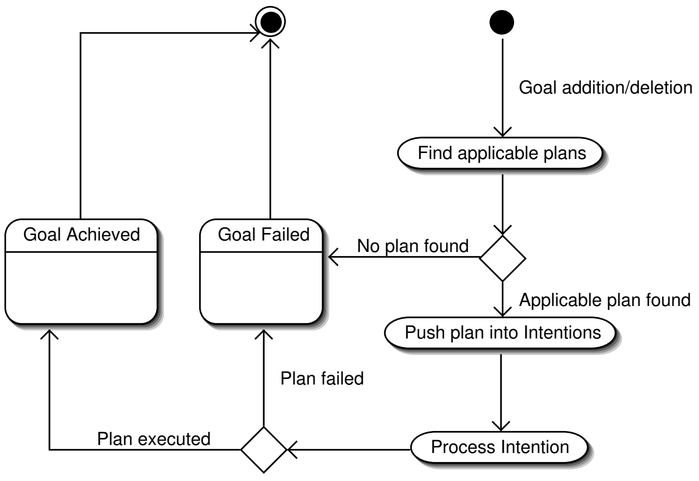


FIGURE 3.8: Simplified AgentSpeak(L) control cycle.

goal addition events to cause the execution of applicable plans *in sequence* [Georgeff and Ingrand, 1989b; Hübner et al., 2006], and the goal to fail only when *all* plans fail, in AgentSpeak(L) goal achievement is an implicit side-effect of a plan being executed successfully.

3.4 AgentSpeak(PL): Planning

Having identified limitations in the AgentSpeak(L) architecture reviewed in Section 3.3 regarding an agent's adaptability to new situations, we conclude that the ability to generate new plans at runtime is needed to address this limitation. Thus, in this section we introduce a planning capability to AgentSpeak(L), calling the resulting system *AgentSpeak(PL)*. However, introducing a procedural plan-based agent language into the inherently declarative formalism of classic planning poses a number of subproblems that we need to address before a concrete system can be realised. We start by analysing the way in which agents are usually designed using AgentSpeak(L), detecting parallels between lower-level AgentSpeak(L) plans and planning operators in Sections 3.4.1 and 3.4.2. The next problem that needs consideration is the more pragmatic one of fitting planning within AgentSpeak's reasoning cycle, which we address in Section 3.4.3. Furthermore, we consider the encoding of planning problem goals within the context of AgentSpeak(L) goals and the invocation of the planner as a result of these AgentSpeak(L) goals in Sections 3.4.4 and 3.4.5. Finally, we deal with the problem of potential failures of plans generated through planning in Section 3.4.6.

3.4.1 Low-level plans versus high-level plans

The design of a traditional AgentSpeak(L) plan library follows a similar approach to programming in procedural languages, in which a designer typically defines fine-grained actions to be the building blocks of more complex operations. These building blocks are then assembled into higher-level procedures to accomplish the main goals of a system. Analogously, an AgentSpeak(L) designer traditionally creates fine-grained actions to be the building blocks of more complex operations, typically defining more than one plan to satisfy the same goal (*i.e.* sharing the same invocation condition), while specifying the situations in which it is applicable through the context part of each plan. For example, an agent that has to move around in London could know many ways of going from one place to another depending on whether a vehicle is available to it, such as by walking or driving a car, as shown in Table 3.10. We call these plans lower-level plans.

```
+!move(A, B) : available(car)
            <- get(car);
               drive(A,B).

+!move(A, B) : not available(car)
            <- walk(A, B).
```

Table 3.10: Movement plans.

High-level plans, on the other hand, use these lower-level plans. For example, if an agent needs to move to a succession of places to accomplish a number of other goals, a high-level plan uses the low-level move plans as illustrated in Table 3.11.

```
+!highLevel : true
  <- !move(home, place1);
     !goal1;
     !move(place1, place2);
     !goal2.
```

Table 3.11: High-level plan that uses the movement plans.

3.4.2 Low-level plans as analogues of STRIPS operators

Now that the details of the planning process and of the AgentSpeak reasoning cycle are clear, we can consider the relation between low-level plans and STRIPS operators in order to establish a link between these two formalisms and allow AgentSpeak plans to be composed into high-level plans through planning. Modelling STRIPS[7] operators to be supplied to a planning algorithm is similar to the definition of the building-block procedures discussed in Section 3.4.1. In both cases, it is important that operators to be used sequentially *fit*. That is, the results from applying one operator should be compatible with the application of the possible subsequent operators, matching the *effects* of one operator to the *preconditions* of the next operator.

[7]PDDL [Fox and Long, 2003] could be used instead, but we use STRIPS for a simpler description. More importantly, AgentSpeak(L) plans do not have conditional effects encoded explicitly, rendering PDDL an overly complex language for our purposes.

Once the building-block procedures are defined, higher-level operations must be specified to fulfil the broader goals of a system by combining these building blocks. In a traditional AgentSpeak(L) plan library, higher-level plans to achieve broader goals contain a series of subgoals to be achieved by the lower-level plans. This construction of higher-level plans that make use of lower-level ones is analogous to the planning performed by a propositional planning system. However, unlike the automated process of planning, the connections between lower-level plans are made only in the mind of the designer.

By doing the *planning themselves*, *designers* must cope with every foreseeable situation an agent might find itself in, and generate higher-level plans combining lower-level tasks accordingly. Moreover, the designer must make sure that the subplans being used do not lead to conflicting situations. This is precisely the responsibility we intend to delegate to a STRIPS planner.

Plans resulting from propositional planning usually consist of a sequence of instantiated operator headers (*e.g.* move(bay1,pigeonHoles);pickup(packet1)), which is not a format directly useable by an AgentSpeak(L) agent. The planning module has to establish the correspondence of operator headers back to AgentSpeak(L) low-level plans and hence convert the STRIPS plan into sequences of AgentSpeak(L) achievement goals to comprise the body of new plans available within an agent's plan library. In this approach, an agent can still have high-level plans pre-defined by the designer, so that routine tasks can be handled exactly as intended. At the same time, if an unforseen situation presents itself to the agent, it has the flexibility of finding novel ways to solve problems, while augmenting the agent's plan library in the process.

3.4.3 Integrating the planner component

Our aim is to provide an extension to AgentSpeak(L) that allows a designer to explicitly specify the world-state that should be achieved by an agent. In order to transform the world to meet the desired state, we need a propositional planner to form high-level plans through the composition of plans already present in its plan library.

This planner component must be integrated into the reasoning cycle somehow so that when the need arises, an agent invokes the planner to generate new plans. Agent interpreters are generally divided between: a kernel containing the basic processes needed for the reasoning cycle, following any theoretical model for an agent architecture; and external capabilities or actions that can be invoked by the kernel and can be implemented by a designer to provide an interface with a particular aspect of the environment. In this light, there are two alternatives to integrating the planner into the reasoning cycle:

- introduce the planning process as part of the interpreter kernel; or

- introduce the planning process as an action within plans.

Introducing the planning process as part of the interpreter kernel implies modifications to the agent interpreter, and entails expanding the interpreter cycle of Figure 3.7 with an additional process; that is, a planning process. Given the potentially high computational cost of planning from first principles, this process must be a last resort option for when a certain event fails to yield applicable plans. In this situation, the planner can be invoked to resolve the impasse.

Introducing the planning process as an action invocable in a regular AgentSpeak(L) plan requires the planning process to be encapsulated in an action implementation. It follows the practice of integrating external capabilities using specialised internal actions [Bordini et al., 2007; Padgham and Winikoff, 2004], in this case a meta-level one. Like in the previous option, planning must be a last resort, and therefore the plans containing the planning action should be positioned after all predefined plans have been attempted.

Prima facie, these options are functionally equivalent, with the second option being simpler in terms of implementation effort. However, using planning as an internal action affords greater flexibility, as a designer can position plans with this action wherever it is most desirable in a plan library. Conversely, if planning is an interpreter-level process, this flexibility can only be achieved through new constructs in the AgentSpeak(L) language indicating where planning should be applied.

Therefore we choose to integrate it through a special AgentSpeak(L) internal action, requiring no change in the language definition. The only assumption we make is the existence of plans that abide by certain restrictions in order to be able to compose higher-level plans taking advantage of planning capabilities introduced in the interpreter.

3.4.4 Goal conjunctions

The notion of planning is closely related to that of goals to be (or declarative goals) [Winikoff et al., 2002]. As planning problems are specified in terms of a desired world-state, it is an inherently declarative form of reasoning. Therefore, the addition of a STRIPS-like planning capability to an AgentSpeak(L)-based architecture requires the consideration of how to link the notion of goals in AgentSpeak(L) to the definition of planning problems.

As we have seen in Section 3.3, plans in AgentSpeak(L) are procedural, and their adoption is a result of either events from the environment or procedural invocations from other plans rather than an explicit desire to achieve a certain world-state. This creates an integration problem, since AgentSpeak goals and traditional planning goals use different notions of goals. We solve this problem by introducing a notation for declarative goals, representing them as regular AgentSpeak(L) goals encoding a conjunction of literals that must be true after a sequence of operators is applied to an initial state. Since goals in this kind of architecture are *procedural*, in that they refer to the execution of *procedural plans* rather than specific world-states to be achieved, we need to define a way to link these two methods of goal description.

Therefore, in addition to the traditional way of encoding goals (to do) for an AgentSpeak(L) agent implicitly as triggering events consisting of achievement goals ($!goal$), we allow desires including multiple beliefs ($b_1, \ldots, b_n$) describing a desired world-state in the form $!goalconj([b_1, \ldots, b_n])$ as goals to be. An agent desire description thus consists of a conjunction of beliefs that an agent wishes to be true simultaneously at a given point in time. Here, the execution of the planner component is triggered by an event $+!goalconj([b_1, \ldots, b_n])$.

Based on this, events of the type $+!goalconj([b_1, \ldots, b_n])$ are used to trigger the execution of a special *planning action*, denoted $plan(G)$, where G is a conjunction of desired goals, through a regular AgentSpeak(L) plan, shown in Table 3.12. This action is bound to a planning component (formally the function of Definition 3.5), and allows the conversion between formalisms to be encapsulated in the action implementation, making it completely transparent to the remainder of the interpreter.

$$+!goalconj([g_1, g_2, \ldots, g_n]) : true \leftarrow plan([g_1, g_2, \ldots, g_n]).$$

Table 3.12: Planner invocation plan.

3.4.5 The planning action

Figure 3.9 illustrates how the internal action to plan takes as an argument the desired world-state, and uses this, along with the current belief database and the plan library, to generate a STRIPS [Fikes and Nilsson, 1971] planning problem. This action then invokes a planning algorithm; if a plan is found, the planning action succeeds, otherwise the planning action fails. If the action successfully yields a plan, it converts the resulting STRIPS plan into a new AgentSpeak(L) plan to be added to the plan library, and immediately triggers the adoption of the new plan.

This representation of goal conjunctions is the key to our approach, as it keeps the principle of procedural execution of plans from AgentSpeak(L) in that the planner itself is just like a regular AgentSpeak(L) plan that is invoked by the agent whenever an event describing the desire to achieve a certain world-state is generated. Furthermore, the plan generated by the planning process is also a regular AgentSpeak(L) plan, invoked by the agent following planner execution, as part of the intention to achieve that particular world-state.

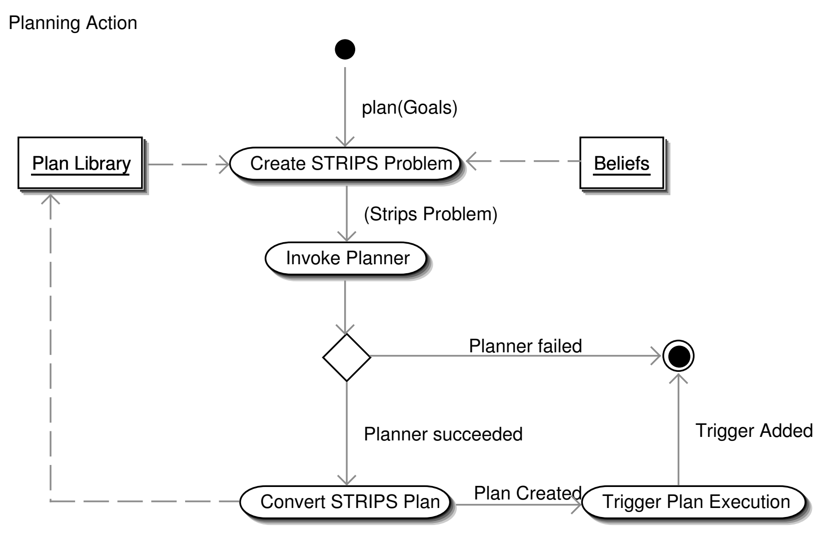


FIGURE 3.9: Operation of the planning action.

3.4.6 Failure in plan execution

As it stands, the achievement of the conjunction of goals is still tied to the first execution of any plan generated by the planner. To address this, the planning action of Figure 3.9 must be redesigned to allow recovery from any potential failure when executing generated plans, and renewed attempts at planning must be made until no new plans can be generated (in line with Figure 3.10).

We thus describe the invocation of the planner as a contingency alternative when all existing plans are proven inadequate to achieve an agent's goals. In this view, when an agent invokes the planner, the planner is expected to either generate a plan that successfully achieves its designated goals or prove that no plan exists to achieve such a goal given the current world-state. However, it is possible that the planner generates a plan which fails to execute to completion due to a number of factors, for example because circumstances change from the moment the plan is created to the moment the plan starts execution. In these types of situation, it may be desirable for an agent to persist attempting to generate a plan to cope with the changing circumstances until the planner too reaches an impasse, as illustrated in Figure 3.10.

In order to accomplish this, it is necessary to use the notion of a *plan-dropping event* introduced by Bordini *et al.* in Jason [Bordini et al., 2007]. In Jason, when a plan adopted as a result of an event +e fails for any reason, an agent receives a plan dropping event −e. For example, if an agent adopts a plan to achieve a certain goal +!goal and this plan fails, it causes the agent to receive an event −!goal signalling the failure. Using this construct, it is possible to recover from a failed plan and reinvoke the planner through the plan shown in Table 3.13, which is a plan analogous to the original *plan to plan* in Table 3.12.

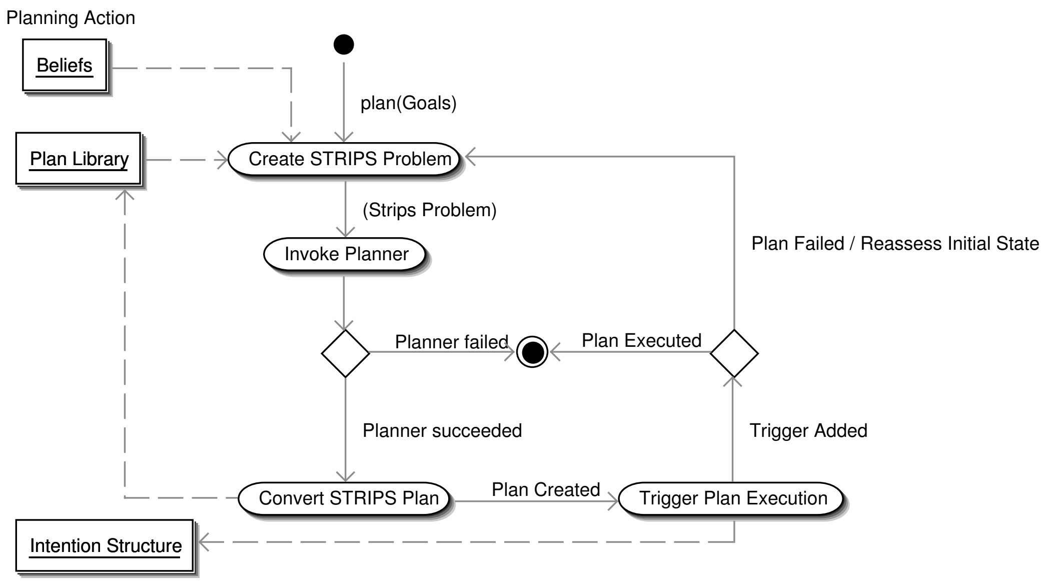


FIGURE 3.10: Operation of the planning action with failure recovery.

$$-!goalconj([g_1, g_2, \ldots, g_n]) : true \leftarrow plan([g_1, g_2, \ldots, g_n]).$$

TABLE 3.13: Plan to handle plan failure.

3.5 From AgentSpeak(L) to STRIPS

Once the need for planning is detected, the plan in Table 3.12 is invoked so that an agent can resort to a planner component. The process of linking an agent to a propositional planning algorithm, illustrated in the diagram of Figure 3.10, includes converting an AgentSpeak(L) plan library into propositional planning operators, declarative goals into goal-state specifications, and the agent beliefs into the initial-state specification for a planning problem. After the planner yields a solution, the ensuing STRIPS plan must be translated into an AgentSpeak(L) plan in which the operators resulting from the planning process become subgoals. In this way, the execution of each operator listed in the STRIPS plan is analogous to the insertion of the AgentSpeak(L) plan that corresponded to that operator when the STRIPS problem was created.

3.5.1 Extracting declarative information

As described in Section 3.4.2, plans in AgentSpeak(L) are represented by a header comprising an invocation condition and a context, as well as a body describing the steps the agent takes when a plan is selected for execution. The invocation condition of an AgentSpeak(L) plan is equivalent to an operator declaration in STRIPS, as it identifies the aim of the plan and lists the relevant variables that are to be instantiated for a concrete invocation to be created. Furthermore the context condition of an AgentSpeak(L) plan is analogous to the preconditions of a STRIPS operator. These correlations can be observed in the move plans of our postman robot example, in particular the plan that contains the action invocation, shown in Table 3.14. In this plan the relevant variables are the starting position and the final position, which must be defined in order to move from one location to the other. Further, the context condition specifies that an agent should only try to move to a different location from its current position, and when its battery is not empty. This correspondence between AgentSpeak(L) plans and STRIPS operators is illustrated in Figure 3.11.

```
+!move(A,B)
  : at(A) & not at(B) & not batt(empty)
  <- move(A,B).
```

Table 3.14: Movement plan from Table 3.8.

The example of Table 3.14 outlines a particularity of AgentSpeak(L) programming, which is that the consequences of actions (such as the new position of a robot after it executes a move action), from the agent's point of view, are implicit, since there is no explicit representation of them in the specification. Therefore, the responsibility of understanding the consequences of executing atomic actions and how they can be used in a plan is delegated to the designer. For an agent to be able to reason about how actions can be chained to achieve broader goals, it must be able to analyse the consequences of these actions and how they fit together. In our example, it is clear that once the agent executes the atomic action to move from A to B successfully, the environment should update the agent's sensor with its new position, causing the agent to revise its belief base so that it no longer believes it is at position A but believes it is at position B.

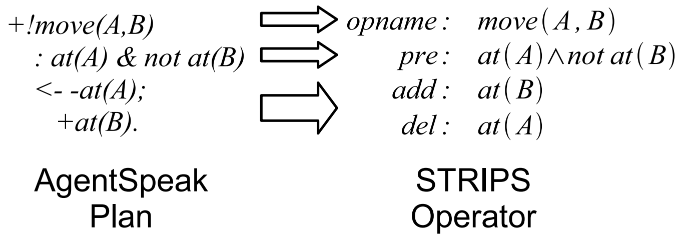


Figure 3.11: Correspondence between AgentSpeak(L) plans and STRIPS operators.

In order to allow our example agent to reason about how the actions available to it affect its broader goals, explicit information regarding the outcome of actions must be included in the building-block plans used by the agent. Therefore, we need to modify the plan in Table 3.14 accordingly, by explicitly encoding in the plan the results of its execution (that the agent is not longer at A and is now at B), as shown in Table 3.15.

```
+!move(A,B)
  : at(A) & not at(B) & not batt(empty)
  <- move(A,B);
     -at(A);
     +at(B).
```

TABLE 3.15: Alternative representation of Table 3.15.

Appropriately, this information about the consequences of a plan is what constitutes the effects of a STRIPS operator, since the added information consists of an explicit list of modifications to the environment, represented as belief additions and deletions. Therefore, we define function *belfilter* to extract belief additions and deletions contained in the body of the plans we intend to convert to STRIPS operators in Definition 3.6.

Definition 3.6 (Belief Update Filter)**.** Let $h_1, \ldots, h_n$ be an AgentSpeak(L) plan body, in which each h_i can be a subgoal or an action, which may be either an atomic action, a subgoal, or a belief update action in the form $+b$ for a belief addition and $-b$ for a belief deletion. We define an abstract function $belfilter(h_1, \ldots, h_n)$ that takes an AgentSpeak(L) plan body and returns the set $B_u = \{h_i | (h_i = +b) \vee (h_i = -b)\}$ of belief update actions in the plan body.

As an example, the application of function *belfilter* to the body of the plan in Table 3.15, consisting of move(A,B), −at(A) and +at(B), results in a set containing only −at(A) and +at(B).

3.5.2 Dealing with atomic actions

Clearly, lower-level plans defined by the designer can (and often will) include the invocation of *actions* intended to generate some effect on the environment. Since the effects of these actions are not usually explicitly specified in AgentSpeak(L) (another example of reasoning delegated to the designer), an agent cannot reason about the consequences of these actions. When designing agents using our technique, we need to explicitly define the consequences of executing a given AgentSpeak(L) plan in terms of belief additions and deletions as well as atomic action invocations. The conversion process can then ignore atomic action invocations when generating a STRIPS specification.

3.5.3 Conversion process

Now that we have all the elements to establish a direct relation between the components of an AgentSpeak(L) plan and a STRIPS operator, we define a generic mapping between them in Definition 3.7.

Definition 3.7 (AgentSpeak(L) to STRIPS mapping)**.** Let $e : b_1 \& \ldots \& b_m \leftarrow h_1; \ldots ; h_n.$ be an AgentSpeak(L) plan, in which e is a triggering event, $b_1, \ldots, b_m$ are belief literals representing the context condition, and $h_1, \ldots, h_n$ are goals or actions, and let $o = \langle pre, post \rangle$ be a generic STRIPS operator with a declaration string o, preconditions pre and postconditions $post$. Further, let $belfilter(h_1, \ldots, h_n)$ be a function that returns only the belief update actions from $h_1, \ldots, h_n$. A new operator can be generated from the AgentSpeak(L) plan where $o \leftarrow e$, $pre \leftarrow b_1, \ldots, b_m$, and $post \leftarrow belfilter(h_1, \ldots, h_n)$.

In our example, we have the following correspondence:

- e is !move(A,B);
- at(A) & **not** at(B) & **not** batt(empty) are belief literals; and
- move(A,B); −at(A); +at(B). is the plan body.

which can be translated into a new STRIPS operator using the mapping of Definition 3.7, yielding the operator shown in Table 3.16.

```
operator move(A,B)
pre: at(A) & not at(B) & not batt(empty)
post: at(B) & not at(A)
```

TABLE 3.16: STRIPS operator created from the plan of Table 3.15.

3.6 From STRIPS to AgentSpeak

As we have seen in the description of the planning action in Section 3.4.5, STRIPS problems generated through the conversion process described in Section 3.5 are processed by a propositional planner. If the planner fails to generate a propositional plan for that conjunction of literals (*i.e.* the goal to be), the planning action fails immediately and the goal is deemed unachievable. If the plan is successful, the resulting propositional plan needs to be converted into an AgentSpeak(L) plan before it can be added to the intention structure.

An AgentSpeak(L) plan is composed of an invocation condition, a context condition and the actual sequence of steps comprising the plan body. These elements must, therefore, be created when converting a STRIPS plan into an AgentSpeak(L) representation. Now, as we have seen, the body of an AgentSpeak(L) plan is analogous to a STRIPS plan, as they both represent the steps that must be taken for a certain goal to be achieved, whereas the other elements (the conditions) are only necessary for retrieving a plan from the plan library. In fact, this use of conditions to identify plans to retrieve from the plan library can be considered a form of *indexing*. Consequently, when generating a plan to be added directly to the intention structure for execution, these indexing elements are not strictly necessary. However, it might be advantageous for an agent to cache newly generated plans for future use instead of having to replan from scratch. In this case, if a plan is to be added to the plan library several issues must be addressed for the correct generation of these indexing elements.

Given these considerations, we proceed to describe two approaches for the generation of AgentSpeak(L) plans from the result of a STRIPS planner. The first approach, described in Section 3.6.1, ignores most issues of integration with an existing plan library, and demonstrates how a STRIPS plan can be easily converted into the body of an AgentSpeak(L) plan; this approach can also be used when the generated plan is not intended to be integrated into the plan library. The second approach, described in Section 3.6.3, builds on the first one, and deals mainly with the generation of context information when a plan is intended for integration with the plan library.

3.6.1 STRIPS actions to plan bodies

A plan from a STRIPS planner is in the form of a sequence $op_1, \dots, op_n$ of operator names and instantiated parameters. This sequence specifies the order in which operators should be executed so that an agent can achieve the desired world-state (according to function *Res* from Definition 3.2), which is functionally equivalent to the execution of plan steps in AgentSpeak. By avoiding the issue of generating a context condition, we can define a new AgentSpeak(L) plan in Table 3.17, where $+!goalconj(Goals)$ is the triggering event that caused the planning module to be invoked.

$$\begin{array}{l} +!goalconj(Goals) : true \\ \qquad\qquad \leftarrow !op_1; \dots; !op_n. \end{array}$$

Table 3.17: AgentSpeak(L) plan generated from a STRIPS plan.

Note that in this approach, we assume that the planning process is faster than the rate of change of the environment, otherwise changes in the environment might endanger the preconditions for the generated plan to be successful. We adopt this assumption here to allow a more generic integration of the planning component to the agent model. The assumption can be dropped, however, by leveraging the work of Ingrand and Despouys [Ingrand and Despouys, 2001], which describes extensions to Graphplan that allow the planning process to be updated as changes in the environment occur.

3.6.2 Generating context information

The addition of the new plan to the intention structure raises the problem of how newly formed plans can be integrated into an agent's existing plan library, or indeed if they should be integrated to the plan library at all. Modifying the plan library at runtime through the addition of new plans effectively changes agent behaviour in at least two ways: first, new plans may cause undesired interactions with the plans that are already part of the plan library, possibly jeopardising the agent's viability in the long term; and second, adding a large number of plans with the same invocation condition may impair the agent's ability to respond in adequate time.

The issue of interference has been discussed in Section 3.3.2, and we have seen that it can be avoided by executing new plans atomically. However, the issue remains of ensuring that plans with the same invocation condition are only selected when their associated plans are most useful. This, in turn, leads to the need for selecting a context condition that maximises the utility of newly generated plans.

In order for a plan to be usefully added to the plan library, the context in which this plan is relevant must be carefully described. If the context is too restrictive, for example by using the entire belief base at the time of planning, the inclusion of a number of irrelevant beliefs will severely limit the future applicability of the new plan. On the other hand, if the context is minimised to include only the preconditions of the first operator, the plan may fail later due to the requirements of subsequent operators.

3.6.3 Adding new plans to the plan library

We have seen that in order for newly generated plans to be added to an AgentSpeak plan library in a way that maximises its future utility to the agent, contextual information should be included in the new plan. Intuitively, the preconditions of any given plan step must have either been made true during the execution of previous plan steps or must have been true from the start of the plan. Therefore, the minimum context condition for any generated plan must specify the preconditions of the first operator, plus the preconditions of any subsequent operators that are not included in the effects of previous operators. We consider this process in more detail below, after presenting Algorithm 1, which describes the generation of such a context condition.

Algorithm 1 uses a Graphplan-like[8] data structure consisting of a directed graph containing alternating levels of propositions and actions. Nodes in an action level are connected to nodes in a proposition level either through precondition edges, denoting that a proposition is a precondition of a given action, or through effect edges, denoting that a proposition is an effect of a given action. Besides the actions included in a planning problem, this

[8]For a more detailed description of Graphplan, see Appendix A.

Algorithm 1 Propagation of preconditions.

```
Require: Plan $\Delta = \{a_1, \dots, a_n\}$, with $n$ steps
Require: Action descriptions $\mathbf{O} = \{\langle a_1, Pre_1, Post_1\rangle, \langle a_n, Pre_n, Post_n\rangle\}$
 1: create a proposition level $P_0$ with no propositions;
 2: for all $a_i \in \Delta$ do
 3:    create an action level $A_i$ containing a node $a_i$;
 4:    add the preconditions of $a_i$ to proposition level $P_{i-1}$;
 5:    connect all $p \in P_i$ to $a_{i-1}$ with precondition edges;
 6:    create a proposition level $P_i$ containing the effects of $a_i$;
 7:    connect all $p \in P_i$ to $a_i$ with effect edges;
 8: end for
 9: for $i = n$ to 1 do
10:    for all $p \in P_{i-1}$ do
11:       if $p$ is not connected to any node in level $A_i$ then
12:          create an action $noop(p)$ in level $A_i$;
13:          connect $noop(p)$ to $p$ through an effect edge;
14:          if $p \notin P_i$ then
15:             create a node $p$ in $P_i$;
16:          end if
17:          connect $p$ to $noop(p)$ with a precondition edge;
18:       end if
19:    end for
20: end for
21: return $P_0$
```

planning graph includes *noop* actions, which connect identical propositions between adjacent proposition levels representing that their truth value remains unchanged between plan steps.

The algorithm initially builds a planning graph populated with the actions of the plan we wish to create a context for, as well as the preconditions and effects of these actions, with edges connecting actions to their preconditions in the previous level, and their effects in the subsequent level. Once the initial graph is generated, proposition levels are iterated backwards and, for each proposition that is connected with a precondition edge to a subsequent action level and not connected with an effect edge to the previous action level, a new *noop* action is created, allowing a proposition to be propagated to the previous proposition level. As the graph is traversed, propositions that are required at one action level are created at the preceding proposition levels until they are either connected to an original action of the plan, or they are propagated through *noop* actions, ensuring that the first proposition level contains all of the preconditions that did not result from the actions in the plan.

To demonstrate the algorithm, consider the postman robot example, and the plan shown in Table 3.6. When the graph for this plan is generated with Algorithm 1, we can see that the only preconditions that are not generated by operators during the plan are $-$at(pigeonHoles), $-$batt(empty), $-$held(packet1), and over(packet1,bay1), which are propagated to the beginning of the graph using *noop* operators. The graph illustrating this run is illustrated in Figure 3.12, with the propagated conditions and their connections shown as dashed nodes. Consequently,

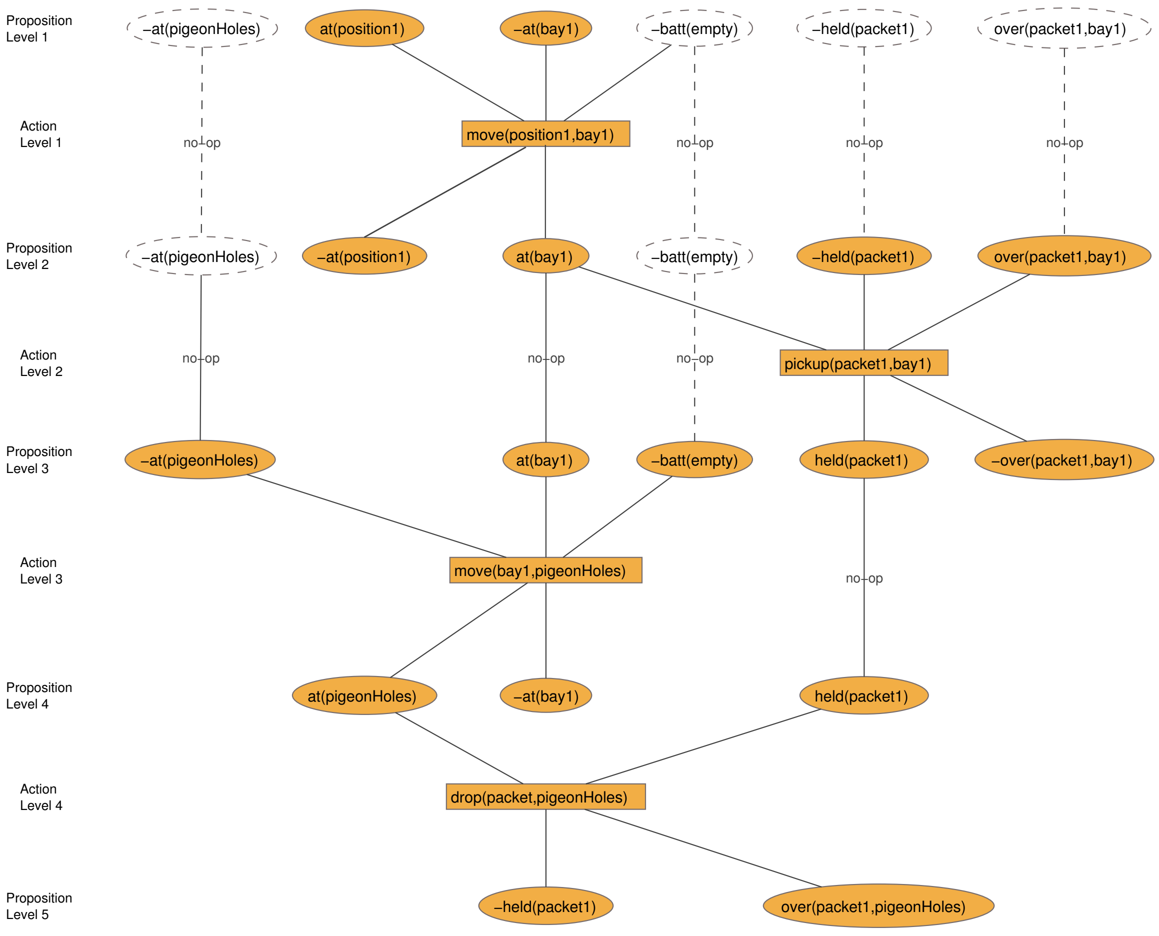


FIGURE 3.12: Graph for the execution of the postman robot example.

the plan generated for the event +!goalconj([over(packet1,pigeonHoles)]) can be added to the plan library with context information. The resulting plan, complete with context condition, is shown in Table 3.18.

```
!goalconj([over(packet1,pigeonHoles)])
    : not at(pigeonHoles) & at(position1) & not at(bay1)
    & not batt(empty) & not held(packet1) & over(packet1,bay1)
    <- !move(position1, bay1);
       !pickup(packet1,bay1);
       !move(bay1,pigeonHoles);
       !drop(packet1,pigeonHoles).
```

TABLE 3.18: Plan with context information.

In terms of computational effort, this algorithm has similar complexity to the graph expansion phase of Graphplan, which has polynomial complexity [Ghallab et al., 2004] in the size of the planning problem for both the size of the graph and the time required to build it. If a plan has m distinct steps, and n distinct propositions, the graph our algorithm creates will have at most $((2 * n) + 1) * m$ nodes, one node for each action and all possible *noops* at each graph level, plus all possible propositions at each proposition level, indicating that the size and time complexity of our algorithm is on the low polynomial scale. Regarding the correctness of the algorithm and its termination guarantee, since the graph building part of the algorithm is a subset of Graphplan [Blum and Furst, 1997], for which a proof of completeness and termination exists, and the rest of the algorithm is an iteration in a directed acyclic graph, it is trivial to show that the algorithm does terminate for any input.

3.6.4 Plan interference

In the ensuing execution of the generated plan, multiple concurrent plans might be stacked in an agent's intentions structure, leading to the possibility of conflicts during their interleaved execution. Plans created *a priori* by a human designer can be manually tailored to avoid any such conflicts. However, when dynamically generated plans are added to the plan library, potential conflicts must be resolved or mitigated somehow. There are multiple ways of addressing this issue, namely:

- delegate the analysis and resolution of conflicting interaction between plans to the designer;
- implement provisions to ensure that the plans used by the planning process are executed atomically;
- drop the entire intention structure before plan adoption and prevent new intentions from being adopted during plan execution; and
- analyse the current intention structure and prospective plan steps during planning to ensure they do not interfere with each other.

Delegating the resolution of concurrency problems to the designer might not be realistic, since the main goal of our work is to diminish the amount of designer tasks, and there is an infinite number of possible combinations of concurrently executing plan steps. On the other hand, analysing the whole intention structure whenever a new plan is added to it involves the introduction of a complex analysis procedure to solve a very limited number of potential conflicts. In this thesis, therefore, we have opted to enable the agent to execute dynamically generated plans atomically (by preventing other intentions from being selected from the stack while a dynamic plan is being executed). We also could have dealt with interference on a corrective basis; that is, if conflicts are expected to arise from the adoption of a new plan, any less important plans in the current intention structure are dropped in favour of the new one. However, the former alternative stems from our goal of minimising the amount of work delegated to the developer, as well as not overloading the reasoning process, potentially sacrificing reactivity.

3.7 Experiments

We have implemented the planning action described in Section 3.4.3 using Jason [Bordini et al., 2005b], which is an open-source Java implementation of AgentSpeak that includes a number of extensions, such as facilities for communication and distribution. In addition to providing an interpreter for the agent language, Jason has an object-oriented API for the development of *actions* available to the agents being developed. Since planning is to be performed as part of a regular AgentSpeak plan, the planning action encapsulates the conversion process of Section 3.5.3 using Jason's *internal actions*.

This implementation was used in a number of toy problems, such as the Blocks World used with the original STRIPS planner [Fikes and Nilsson, 1971], as well as some examples from the AgentSpeak literature [Rao, 1996]. Solutions for these problems were created using both a procedural approach characteristic of traditional AgentSpeak agents, and a declarative one, in which high-level plans are omitted entirely and left to be derived by the planning system. We provide three experiments: a concrete production cell with multiple combinations of usage in its processing units and how it can be described using a planning capable agent in Section 3.7.1, as well as an extension of this experiment showing the impact of plan reuse in runtime performance in Section 3.7.2, and finally an abstract experiment used to gather performance data on the computational effort gap between a planning and a non-planning agent in Section 3.7.3.

3.7.1 Production cell example

Planning, in general, is a recognisably complex problem, hence it is important to evaluate the performance penalty incurred by the use of state-space planning during the reasoning process.

As a result, we have developed a production cell scenario, in which parts must be processed by different processing units, depending on the type of part. Parts enter the production cell for processing through the feed belt and, once processed by all the appropriate processing units, they are removed from the cell through the deposit belt. Every processing unit is responsible for performing a different kind of operation on the part being processed, and can process only one part at a given moment. The general layout of this production cell is illustrated in Figure 3.13, taken from [Meneguzzi et al., 2004].

Now, each part introduced into the cell can be processed by one or more processing units, as determined by the type of component being processed. The control of the production cell is entrusted to two different BDI agents using different types of AgentSpeak interpreter, one using AgentSpeak(L) static plans developed for each possible situation in our experiment,

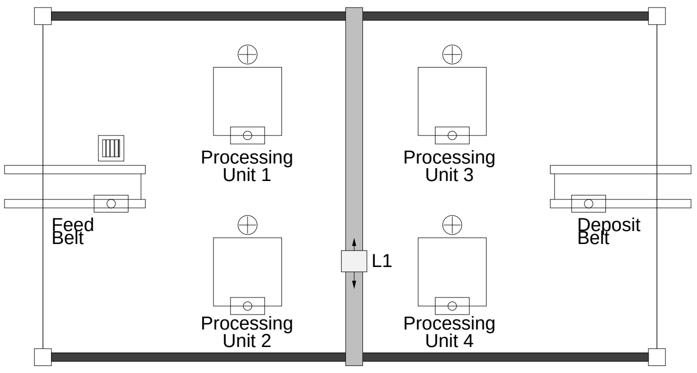


Figure 3.13: Overview of the production cell.

and another using AgentSpeak(PL) planning capabilities to schedule the work of the production cell through its newly created plans. For testing purposes, we consider a production cell with four processing units, in which parts of three different types can be processed:

- Type One must be processed by Processing Units 1, 2 and 3;
- Type Two must be processed by Processing Units 2 and 4; and
- Type Three must be processed by Processing Units 1 and 3.

The experiment consists of simulating the arrival of parts of random types in a production cell, once controlled by a traditional AgentSpeak(L) agent and once controlled by an AgentSpeak(PL) agent. The original AgentSpeak(L) constitutes our baseline for the experiment, since it has zero overhead due to planning. The time spent planning and achieving the final processing of every part was measured for each agent for an increasing number of parts, ranging from 10 to 100 in 10 part increments, and repeated 10 times to amortise fluctuations in the underlying hardware and operating system the results of which can be seen in the graph of Figure 3.14.

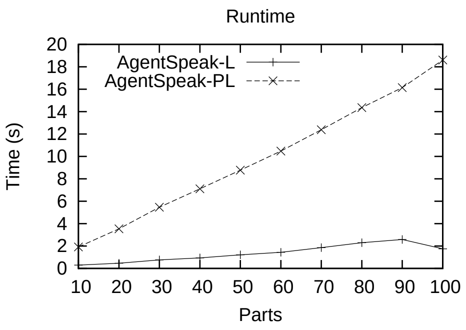


Figure 3.14: Comparison of running times for Production Cell scenario.

In this graph we can see that, though the planning version takes significantly more time to perform its reasoning cycle, time overhead increases roughly linearly with the number of parts, as the time spent in planning accumulates. This is, however, an expected result, since planning from first principles is much more computationally expensive than the simple unification process normally required of AgentSpeak(L) to match existing plans. However, traditional AgentSpeak(L) cannot deal with situations for which no plans exist, which is the main advantage of our approach. It is interesting to note that in this scenario no plan reuse strategy was used, so these results can easily be improved using the reuse technique of Section 3.6.3, as we see next.

3.7.2 Impact of plan reuse

In the above experiment, we saw that planning from first principles is a costly endeavour and needs to be amortised through a plan reuse strategy, as described in Section 3.6.3. To assess the impact of reusing plans in compensating for the computational effort spent in planning we have developed a variant of the experiment of Section 3.7.1. This experiment consists of simulating the arrival of parts of three types in three production cells, one controlled by a traditional AgentSpeak(L) agent (AS), another controlled by a *naive* version of AgentSpeak(PL) (NaiveAS) that does not reuse plans and one controlled by the *complete* AgentSpeak(PL) (ASPL) capable of reusing plans. Here, whenever a new part arrives for processing at the cell controlled by NaiveAS, the full planning process is invoked to generate a new plan, regardless of previous instances of the same problem having been considered in the past. The time spent planning and achieving the final processing of every part is measured for each agent for an increasing number of parts, ranging from 10 to 100 in 10 part increments.

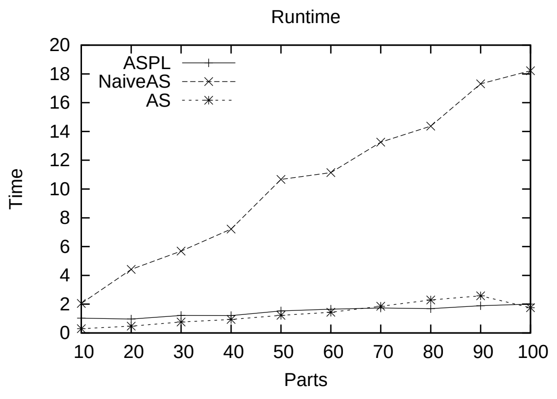


FIGURE 3.15: Running times for the Production Cell scenario.

The results of this experiment can be seen in the graph of Figure 3.15, which shows that, though NaiveAS takes significantly more time to perform its reasoning cycle, this overhead is constant. Now, when the plan reuse strategy is used by ASPL, runtime performance

	AS	**ASPL**
# plans	12	7

Table 3.19: Plan library size comparison.

improves considerably, approaching that of AS. With three different part types, the number of possible world configurations at the time of planning is limited, and most of the planning effort occurs at the beginning of the agent execution. As more parts of the same type are introduced in the production cell, the plans generated previously are invoked rather than the planning module, *amortising* the cost of the initial planning. Evidence of this effect is provided by the ASPL curve approaching that of AS as the number of total parts increases. Moreover, since the plans generated through planning are a linear sequence of actions, which do not rely on the tests distributed throughout a branching structure of plans in the plan library, they are inherently faster to be executed than the equivalent AS representation, surpassing it in the long term.

It is important to note that, although ASPL can create plans for situations in which AS would fail, we have avoided using these problems in our benchmark, focusing only on run-time, by considering an AS agent with plans for all situations possible during testing. By relying on a planning approach, we also diminish the size of the agent specification, since we no longer need to create a procedural plan to cope with every world configuration relevant to the accomplishment of an individual plan. The numbers of plans necessary in the (initial) plan libraries are shown in Table 3.19.

3.7.3 Abstract example

One of the main advantages of specifying behaviour in a declarative manner is a more concise specification. In order to demonstrate this, we designed an abstract scenario that shows how an agent can be specified more concisely using an AgentSpeak(PL)-based declarative specification than a traditional AgentSpeak(L) specification. The scenario is representative of a number of real-world situations, for example a sandwich bar, in which the large number of combinations of fillings for sandwiches makes it very hard to enumerate all combinations, as well as the detailed plans for making these sandwiches. Another possibility is in the assembly of new cars or computers, which use similar chassis but result in quite different models depending on the parts that are mounted onto them. We keep this scenario abstract to focus on the performance numbers.

Specifications in the scenario follow a traditional AgentSpeak(L) approach consisting of defining low-level plans that achieve parts of higher-level goals, as well as the main *event reaction* plans that define which high-level objectives need to be achieved when certain events are perceived in the environment. The comparison focuses on the definition of plans to achieve high-level goals in response to events in the environment, and demonstrates that using a procedural approach requires a larger plan-library in order to cope with all possible combinations of low-level objectives that may need to be accomplished in pursuit of high-level ones.

```
+!action1 : have(resource1)
        <- +goal1.

+!action2 : have(resource2)
        <- +goal2.

+!action3 : have(resource3)
        <- +goal3.

+!action4 : have(resource4)
        <- +goal4.

+!getResource(Resource) : not have(Resource)
        <- +have(Resource).
```

TABLE 3.20: Low-level plans for the abstract scenario.

In this particular scenario, and common to both agent specifications, we consider four low-level *goals* that can be achieved through corresponding low-level plans, named goal1, goal2, goal3, and goal4. These low-level plans require a certain resource in order to accomplish their goals, so a plan to obtain these resources is also included in the library of low-level plans. The set of low-level plans used in this experiment is shown in Table 3.20.

Our experiment focuses on evaluating the runtime overhead of performing planning at runtime to allow for a more concise plan library based on declarative goals. Here, high-level goals require the achievement of one or more low-level goals, and a procedural specification requires the steps to achieve each low-level goal and its pre-requisites to be stated explicitly, as illustrated in Table 3.23. This specification must include contingency plans for every possible environment configuration in order to ensure that the right low-level plan is invoked. This level of detail contrasts with the declarative specification shown in Table 3.22, which relies on the underlying planner to select the necessary low-level plans.

Event	**High-level goal**
event1	goal1
event2	goal2
event3	goal1, goal2
event4	goal3
event5	goal1, goal3
event6	goal1, goal2, goal3
event7	goal4
event8	goal1, goal4
event9	goal2, goal4
event10	goal1, goal2, goal4

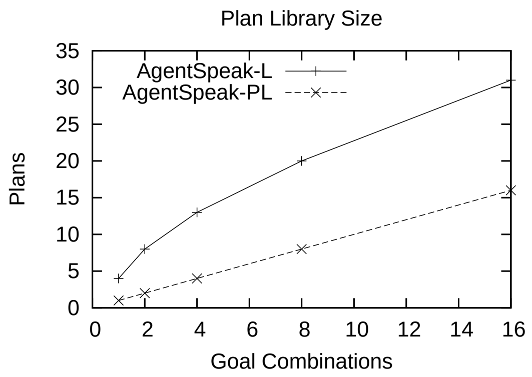
TABLE 3.21: High-level goals for the abstract scenario.

```
+!goalConj(Goals) : true
        <- .plan(Goals,true).

+event1 : true
        <- !goalConj([goal1]).
```

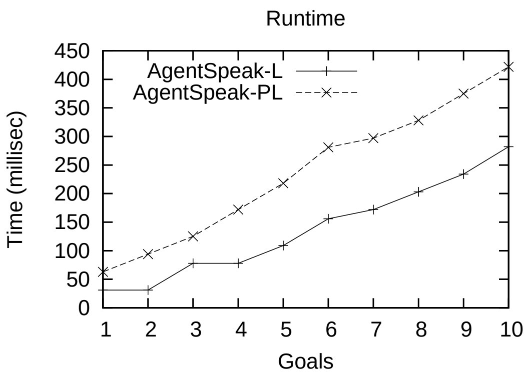
TABLE 3.22: High-level plans for AgentSpeak(PL).

```
+!goal1 : goal1
        <- true.

+!goal1 : have(resource1) & not goal1
        <-  !action1.

+!goal1 : not have(resource1) & not goal1
        <-  !getResource(resource1);
        !action1.

+event1 : true
        <- !goal1.
```

TABLE 3.23: High-level plans for AgentSpeak(L).



(a) Plan library sizes.



(b) Processing times.

FIGURE 3.16: Graphs for the abstract scenario.

As the number of potential subgoals increases, the size of a traditional AgentSpeak(L) specification increases not only with the specification of the high-level plans, but also with the contingency plans, whereas an AgentSpeak(PL) specification increases linearly, as illustrated in the graph of Figure 3.16(a). For the runtime evaluation, we considered ten possible combinations of the four low-level goals defined for this scenario, enumerated in Table 3.21. One agent of each type (procedural and declarative) was then deployed in a simulated environment having to cope with an increasingly large number of the events of Table 3.21 and the time spent achieving the subgoals was measured, the results of which are illustrated in Figure 3.16(b).

The results of the graphs in Figure 3.16 show that as the combinations of goals increase in size, the number of plans needed to achieve them in a traditional plan library increases at a much faster rate than in AgentSpeak(PL). The ability to create new plans for each combination of goals at runtime means that a designer does not need to specify a plan for each combination of goals. Furthermore, the time spent planning to create these new plans imposes an overhead at runtime, but as the graph of Figure 3.16(b) indicates, this overhead increases at roughly the same rate as the time spent selecting preexisting plans in the larger AgentSpeak(L) plan library.

3.8 Related Work

In this section we examine some of the efforts that are somehow related to our work. We start with architectures that use declarative goals, which is one of the contributions of our work, following with work related to the derivation of context conditions in Section 3.8.6. Work on the declarative nature of goals as a means to achieve greater autonomy for an agent is being pursued by a number of researchers. Here we consider the approaches to declarative goals currently being investigated, namely those of Hübner *et al.* (Jason) [Hübner et al., 2006], van Riemsdijk *et al.* [van Riemsdijk et al., 2005] and Meneguzzi *et al.* [Meneguzzi et al., 2004]. There are multiple interpretations as to the requirements and properties of declarative goals for an agent interpreter, and while some models consist of an agent that performs planning from first principles whenever a goal is selected, others argue that the only crucial aspect of an architecture that handles declarative goals is the specification of target world-states that can be reached using the traditional procedural approach. Besides the issue of how declarative goals can be incorporated into practical architectures, other researchers have investigated the issue of using planning modules to augment existing architectures, such as in Propice-Plan [Ingrand and Despouys, 2001] and JADEX [Sardiña et al., 2006; Walczak et al., 2006]. These efforts provide insight into many practical issues that may arise from the integration of BDI with AI planners, such as how to modify a planning algorithm to cope with changes in the initial state during planning [Ingrand and Despouys, 2001], and how to cope with conflicts in concurrently executing plans [Walczak et al., 2006].

3.8.1 Propice-Plan

Propice-Plan [Ingrand and Despouys, 2001] is a PRS-based system that includes planning capabilities through a modified version of the IPP planner [Köhler, 1998]. It includes refinements to allow an agent to anticipate alternative execution paths for its plans, as well as the ability to update the state of the planning process in order to cope with a highly dynamic world. Propice-plan is very similar in principle to our AgentSpeak(PL) architecture described here, but it differs in several key aspects, such as its reliance on a modified PRS description formalism for agents as well as relying on a tailor-made planner implementation, limiting the choice of planners to be used in tandem with the agent interpreter.

3.8.2 Jason

The notion of declarative goals for AgentSpeak that takes advantage of the context part of the plans (representing the moment an implicit goal becomes relevant) was introduced by Hübner *et al.* [Hübner et al., 2006], and implemented in Jason [Bordini et al., 2005b], which is the same AgentSpeak interpreter we use in our AgentSpeak(PL). More specifically, plans that share the same invocation condition refer to the achievement of the same goal, so that a goal can only be considered impossible for a given agent if all plans with the same invocation condition have been attempted and failed. In Jason, these plans are modified so that the last action of every plan consists of testing for the fulfilment of the declared goal, and then the plans are grouped and executed in sequence until one finishes successfully. A plan only succeeds if at the end of its execution an agent can verify that its intended goal has been achieved. This approach retains the explicitly procedural approach to agent operation (a pre-compiled plan library describing sequences of steps that an agent can perform to accomplish its goals), but adding a more robust layer for handling plan-failure.

3.8.3 GOAL, Dribble and their extensions

Several researchers have worked on a family of declarative agent languages (including GOAL, Dribble and their extensions) and investigated possible semantics for these languages [Hindriks et al., 2001; van Riemsdijk et al., 2005]. All of these languages have in common the notion that an agent is defined in terms of beliefs, goals and capabilities, which are interpreted in such a way as to select and apply capabilities in order to fulfil an agent's goals. These approaches have evolved from GOAL [Hindriks et al., 2001] into a declarative semantics very similar to that of X-BDI [Móra et al., 1999], in which an agent's desires express *world-states* that must be achieved by the selection and application of capabilities.

3.8.4 Planning in JADEX

The work of Walczak *et al.* [Walczak et al., 2006] is another recent approach to merging BDI reasoning with planning capabilities, and is based on a continuous planning and execution framework implemented in the JADEX agent framework [Pokahr et al., 2005b]. The system uses a modified HTN state-based planner which uses domain-specific information to select the actions to achieve goals or refine goals in an agent's agenda. The emphasis in this system is on performance and reaction time rather than generality, since JADEX uses a Java-like representation for the agent's data structures, such as goals and actions.

3.8.5 HTN planning in BDI

Considering the many similarities between BDI programming languages and HTN planning, Sardiña *et al.* [Sardiña et al., 2006] formally define how HTN planners can be integrated into a BDI architecture. In this work, Sardiña shows that the HTN process of systematically substituting higher-level goal tasks until concrete actions are derived is analogous to the way in which a PRS-based interpreter pushes new plans into the intention structure, replacing an achievement goal with an instantiated plan. Taking advantage of this almost direct correspondence, an HTN planner is used to add *lookahead* capabilities to an agent, allowing it to optimise plan selection and maximise an agent's chance of successfully achieving goals. By verifying beforehand the selection of plans for achieving subgoals, the agent minimises the chance of failure as a result of poor plan selection.

The AgentSpeak(L) style, two-tiered approach to the definition of planning problems is present in HTN planning and the similarities with a BDI interpreter are significant, both in terms of advantages and limitations. Just as in an AgentSpeak intention structure, plans generated by an HTN planner are limited by the set of substitution methods available and constraints imposed on composite tasks. The relation between BDI agents and HTN planners is explored by Sardiña *et al.* [Sardiña et al., 2006], and their research outlines the fact that HTN planners can be employed by an agent to decide which plans to instantiate in order to succeed, but does not allow the agent to create new plan structures.

3.8.6 Regression of Web Services

The idea of analysing one formalism to derive planning-like pre and post conditions has been attempted previously in the context of web service composition through planning. Initial efforts by McIlraith and Fadel [McIlraith and Fadel, 2002] at a theoretical level, involved converting web services described by hand using Golog into PDDL and ADL. However, this lacked generality due to its heavy reliance on human intervention in the process, preventing it from being used in a completely automated fashion, as is needed by our work. Later, this idea was refined by Pistore *et al.* [Pistore et al., 2005], converting web services defined in BPEL4WS into PDDL, allowing for automation. However, BPEL is much more complex than AgentSpeak(L), and understandably the conversion algorithm has polynomial complexity, though on the *exponential* scale. In this respect, our approach compares favourably by having non-exponential polynomial complexity.

3.8.7 Comparison and Discussion

In addition to the models described in this section, variations of the way an agent interpreter handles declarative goals have also been described. These approaches advocate the use of fast propositional planners to verify the existence of a sequence of actions that fulfil a declarative goal [Meneguzzi et al., 2004]. The planning process in this setting allows the consideration of the entire set of available operators to create new plans, providing a degree of flexibility to the agent's behaviour.

The approaches in Jason, GOAL and Dribble deal with important aspects of declarative goals in agent systems, such as the verification of accomplishment and logical properties of such systems. However, support for declarative goals in Jason still requires a designer to specify high-level plans, while the formalisms described by van Riemsdijk lack any analysis of the practicality of their implementation. Jason, in particular, is more concerned with exhaustively trying all high-level plans sequentially to accomplish a goal.

The addition of a planning component to a BDI agent model has recently been revisited by other researchers, especially by Sardiña *et al.* [Sardiña et al., 2006] and Walczak *et al.* [Walczak et al., 2006]. The former describes a BDI programming language that incorporates Hierarchical Task Networks (HTN) planning by exploring the similarities between these two formalisms, but agents in this approach are no more flexible than they would normally be, since they still rely on the same plans originally designed prior to deployment. The latter approach is based on a specially adapted planner to support the agent, preventing the model from taking advantage of novel approaches to planning.

3.9 Conclusions

Recent approaches to the programming of agents based on declarative goals rely on mechanisms of plan selection and verification. However, we argue that a declarative model of agent programming must include not only constructs for verifying the accomplishment of an explicit world-state (which is an important capability in any declarative agent), but also a way in which an agent designer can specify *only* the world-states the agent has to achieve and the description of atomic operators allowing an underlying *engine* to derive plans at runtime. In this chapter we have argued that propositional planning can provide one such engine, drawing on agent descriptions that include atomic actions and desired states, and leaving the derivation of actual plans for the agent at runtime.

As a consequence, we have developed an architecture for planning agents, which we used to explore how the addition of a planning component can augment the capabilities of a plan library-based agent. In order to exploit the planning capability, the agent uses a special planning action to create high-level plans by composing new plans within an agent's plan library. This assumes no modification to the AgentSpeak language, and allows an agent to be defined so that *built-in* plans can still be defined for common tasks, while allowing for a degree of flexibility for the agent to act in unforeseen situations. Moreover, despite the inherent complexity of planning from first principles, we have addressed the increased computation cost of planning through a plan reuse technique.

We have created a prototype to test our approach by extending an open source AgentSpeak(L) interpreter. The prototype has been empirically tested on a number of scenarios, allowing us to evaluate how the planning ability impacts the runtime performance of an agent, as well as the size of a plan library. Moving from a traditional AgentSpeak design method to one based on dynamically generated plans results in a reduction of the plan description size, as it is no longer necessary to enumerate relevant combinations of lower-level plans for the agent to be able to react to different situations.

This system can be improved in a number of ways in order to better exploit the underlying planner component. For example, the effort spent on planning can be moderated by a quantitative model of control, so that an agent can decide to spend a set amount of computational effort into the planning process before it concludes the goal is not worth pursuing. This could be implemented by changing the definition of the triggering condition composed of a conjunction of literals used in our model to include a representation of a motivational model, which can be used to tune the planner and set hard limits to the amount of planning effort devoted to achieving that specific desire.

Our system can also be viewed as a way to extend the declarative goal semantics proposed by Hübner *et al.* [Hübner et al., 2006], in that it allows an agent designer to specify only desired world-states and basic capabilities, relying on the planning component to form plans at runtime. Even though the idea of translating BDI states into STRIPS problems is not new [Meneguzzi et al., 2004], our idea of an encapsulated planning action allows the usage of any other planning formalism sufficiently compatible with the BDI model.

Chapter 4

Motivations in meta-reasoning

As we have seen, flexible agents need the ability to create new plans at runtime, allowing them to overcome situations not covered by their original plan library. However, as a plan library grows in size and complexity, the decision to adopt specific courses of action from among multiple possibilities ceases to be as trivial as choosing the first plan that matches a certain condition. Since autonomous agents are expected to have control over their internal state and behaviour [Jennings, 2000] to be able to act effectively in a complex environment, an agent must be capable of making this decision without direct human support. In turn, control over an agent's internal state requires reasoning about the reasoning process itself, in what is commonly understood as meta-level reasoning (or meta-reasoning). However, most agent architectures include plan libraries that contain only *action-directed* plans invoked through a simple trigger-response mechanism. Thus, if there is a possibility of conflict between any two plans in the plan library, a designer must make sure that these conflicts are handled through extra steps within the plans themselves. As a consequence, the function of meta-reasoning is not explicit, but mixed with the action-directed plans, and the ensuing agent does not handle any conflicts that were not foreseen by the designer.

Meta-reasoning enables an agent to explicitly consider goals before committing to their achievement, as well as consider courses of action before executing plans, as opposed to simply reacting to events in the environment. However, research on agent architectures traditionally focuses instead on achieving quicker reactions to events in the environment. In traditional architectures, the lack of meta-reasoning at runtime is addressed by the developer foreseeing any contingencies in an agent's capabilities, which may not be realistic in many complex environments.

It is certainly possible to develop agents with meta-level reasoning capabilities using existing architectures, but the absence of a distinct component responsible for meta-level control increases the complexity of an agent's specification while limiting its runtime flexibility, since the function of meta-reasoning must then be accomplished by extra steps within an agent's action-directed plans. Therefore, we believe the development of effective autonomous agents can be facilitated by the inclusion of an explicit meta-reasoning capability. Though existing efforts in adding meta-reasoning strategies to agent architectures illustrate the possibilities of such a module, these efforts have lacked generality, often restricting meta-reasoning to specific domains [Raja and Lesser, 2004] in which meta-reasoning is *hard-coded* into an agent architecture. We believe that a suitable abstraction for this component is needed, and the body of work on motivated agency (as reviewed in Chapter 2) provides a strong basis for meta-level reasoning. Research on motivated behaviour has been conducted in areas such as psychology (*e.g.* [Morignot and Hayes-Roth, 1996]), ethology (*e.g.* [Balkenius, 1993]), and philosophy (*e.g.* [Mele, 2003]), including some work in computer science (*e.g.* [Cañamero, 1997; Grand and Cliff, 1998; Luck et al., 2003; Norman et al., 2004]).

In this chapter, we develop a meta-level reasoning component based on the notion of motivation. Meta-reasoning is specified through a motivation-oriented language with functions for updating the motivational state of an agent as well as adopting and dropping goals as a result of this motivational state. Our adoption of the motivation abstraction follows the the extensive previous work on motivations described in Section 2.4, as well as in keeping with the folk-psychology orientation of the BDI architecture. We start the chapter by discussing the reasons for adding meta-reasoning to an agent architecture in Section 4.1, followed by an abstract model of motivations in Section 4.2. We take a concrete model of motivations that closely matches our abstract model in Section 4.3, and extend it in Section 4.4. We then integrate this extended model into an agent interpreter in Section 4.6, and test its effectiveness in the experiment of Section 4.7. Finally, we conclude the chapter with a brief discussion in Section 4.8.

4.1 Reasons for Meta-reasoning

Several efforts towards refining the *deliberation* process (or reasoning), which can be viewed as meta-level reasoning, have been proposed recently. For example, meta-reasoning[1] can be used to optimise task scheduling using some utility measure [Raja and Lesser, 2004], and to improve and manage concurrent goal execution by exploiting opportunities and avoiding conflicts [Pokahr et al., 2005a; Thangarajah et al., 2003b]. Most of these strategies rely on optimising agent behaviour by comparing some intrinsic characteristic of an agent's plans, execution time, or some abstract notion of utility.

For example, the widely known and used BDI architectures are usually of two types — procedural and declarative — and in both cases, there are advantages of meta-reasoning. *Procedural* architectures require detailed plans and triggers to be described by the designer, hence conflicts between plans must be foreseen, with conflict resolution strategies embedded in each procedural plan as extra *management* steps. For example, if two procedural plans require exclusive access to a particular resource, a designer must make sure that each plan

[1]We use meta-reasoning and meta-level reasoning interchangeably.

checks whether the other plan is accessing the resource before trying to use it. The function of these extra steps is analogous to meta-reasoning, since they are not action-directed, but rather are evaluating the executing agent's internal state in querying what other plans are being executed. Thus, in procedural languages, specifying meta-reasoning separately from the plans removes the need to replicate such internal *management* steps throughout the plan library, facilitating development. Alternatively, *declarative* architectures are defined by desired states to be achieved and capabilities with which an agent can achieve them, where an interpreter selects capabilities to achieve goals, and conflict resolution must be done by this interpreter. In declarative languages, the lack of some goal selection policy means that goals and plans are selected arbitrarily, since in theory the designer does not specify precisely *how* goals are to be achieved.

The generic definition of meta-level reasoning as reasoning about the reasoning process is rather broad, so we must define a narrower scope for meta-level processing in order to apply it to any concrete architecture. While there are many different models for inclusion of meta-reasoning into an agent architecture, we have already seen (in Chapter 2), that *motivation* provides one valuable way to do so, and at the same time offers a meaningful abstraction that is useful for understanding the meta-reasoning process (as humans), and for modelling the meta-reasoning of other agents. Moreover, motivation has already been used in the kind of BDI architecture we are considering, albeit in a more limited fashion, with motivations hard-coded and used to drive procedural goals. Therefore, while we adopt the basic motivation abstraction used in previous BDI architectures, we must adapt it to the needs of the architecture defined in Chapter 3. More specifically, the resulting model needs to:

- associate motivations with the generation of declarative (*to be*) goals;
- use motivational intensity to select and prioritise intentions adopted to achieve the most rewarding goals;
- mitigate motivations based on the fulfilment of *declarative* goals; and
- avoid the need to hard code motivations in the agent architecture.

Now, since the steps of the reasoning cycle of our AgentSpeak(PL) architecture are very similar to those of traditional BDI architectures, and the functions provided by some of the models of motivations in the architectures of Section 2.4 were designed to be integrated into this kind of architecture, it is possible to leverage them in our modified model of motivations. In these architectures, motivations underpin the rational adoption of goals, allowing the comparison and prioritisation of competing goals. As a consequence, we can use these functions for goal generation and prioritisation with little modification.

As we have seen, an agent's perceptions are determined by its environment and captured by its beliefs. In turn, the intensity of an agent's motivations varies through time as a function of its beliefs. Accordingly, the architectures surveyed in Section 2.4 typically provide some kind of function that associates a motivational value representing intensity with world-states and actions. As far as the model of motivation is concerned, a motivation's intensity serves two purposes: first to determine the relative importance of a motivation compared to others, and second to determine the point at which an agent is sufficiently motivated to generate a goal and actively try to mitigate this intention. In terms of meta-level control, intensity information indicates how important a certain world-state is, which allows an agent to anticipate the motivational effect of certain courses of action before committing to them, allowing it to select plans more effectively.

In most motivated agent architectures, the associations between motivations and goals that mitigate them, and between specific world-states and motivational intensity updates, are hard-coded within a belief update function, making it difficult to use the architecture in different domains. In order to use motivated architectures for *generic* agent programming, therefore, it is important for the definition of motivations and their dynamics to be specified outside the agent architecture. For example, the association of specific world-states in an agent architecture to motivational intensity updates requires some appropriate representation, and doing this in a generic and reusable fashion requires a specification language.

In summary, we use motivated reasoning to perform three functions: reasoning about goal adoption, goal prioritisation, and plan selection. Translating these functionalities to the BDI model requires associating motivational intensity to the belief database, and modifying the reasoning process responsible for committing to intentions so that it uses motivational information.

4.2 An abstract model of motivation

When considering the use of motivations as an abstraction for meta-reasoning, we must take into account the operations that this type of reasoning performs so that the model of resulting motivations adequately captures the desired properties of meta-level reasoning. As we have seen in Section 2.4, meta-reasoning operates on the data structures of the underlying agent architecture, which in the case of BDI, are beliefs, desires and intentions.

In particular, we want to use motivations for three key purposes: first to choose between goals; second to assess goal achievement; and finally, to assess motivation mitigation. In this context, we want to use a model of meta-reasoning to select the goals that provide the best rewards for an agent; thus one particular aspect of motivations we want to capture is goal selection. The idea of selecting goals to achieve a reward leads to the need for modelling motivations in such a way that an agent's reasoning cycle can select between goals. Moreover, after goals are adopted, an agent needs to determine when these goals have been achieved satisfactorily, so that we need an assessment of the degree to which goals have satisfied a certain motivation. In order to provide these properties in our own model, therefore, we need to consider how such an assessment of the importance of certain world-states can be undertaken, so that goals can be selected to achieve more important world-states. Finally, after goals have been selected, the model must offer some means to determine when, and to what extent, achieved goals mitigate their associated motivation.

We must not overlook the fact that the main reason we propose motivations as an abstraction for meta-reasoning is that it allows a designer to model the rewards of certain behaviours even when the domain does not provide a natural way of assessing these rewards objectively. Thus, our model of motivations must not only provide the properties described above, but must also allow motivations to be custom-defined alongside the agent they help to control.

Now, from our previous survey of computational models of motivated behaviour, it is clear that while there are several relevant pieces of work, the existing models lack generality in that most are *hard-coded* within *ad hoc* architectures developed exclusively for the simulation of some type of artificial ecosystem, and are thus unsuitable for specifying meta-level behaviour. Nevertheless, some of these models provide some indications of the way in which we might construct our generic motivations-based mechanism for meta-reasoning. In particular, the mBDI model developed by Griffiths and Luck [Griffiths and Luck, 2003] includes abstract mechanisms for intensity evaluation and goal generation, and is notably compatible with the interpretation of the BDI architecture we consider in this thesis, more specifically that of AgentSpeak(L). As a result, we can leverage this model of motivations in our work, and thus use mBDI as the starting point for our abstract model of motivations.

However, this model suffers from two major limitations. First, it does not consider the notion of goals *to be* introduced into AgentSpeak(L) in Chapter 3 in that it only evaluates motivational reward in relation to plan execution rather than achieved world-states. Consequently, we need to adapt it to handle the adoption and achievement of goals *to be*, with the model and its adaptations being presented in Sections 4.3 and 4.4. Second, the mBDI model only specifies particular key functions in an *ad hoc* formalisation that seems to bear no relation to the manner of its development. This is important, for one of the aims of our work is to provide agent languages that facilitate the programming of agent systems in a generic and reusable fashion. In consequence, we argue that there should be a motivation description language, increasing applicability for the design of meta-reasoning strategies. We develop such a motivation description language, described in Section 4.5.

4.3 Griffiths's mBDI model

As we have seen, we need a model of motivations that includes a minimum set of functions that allow: the evaluation of world-states in terms of motivational value; the generation of goals that best satisfy a motivation; and the identification of how motivations are mitigated as a result of goals being achieved. Based on our survey of computational models of motivation the most easily adaptable model of motivations that is compatible with the BDI architecture is the one from Griffiths and Luck [Griffiths and Luck, 2003], which contains these functions albeit not catering for the notion of goals *to be.* According to Griffiths and Luck [Griffiths and Luck, 2003], a motivation is a tuple $< m, i, t, f_i, f_g, f_m >$, where m is the motivation name, i is its current intensity, t is a threshold, f_i is an *intensity update function*, f_g is a *goal generation function*, and f_m is a *mitigation function.*

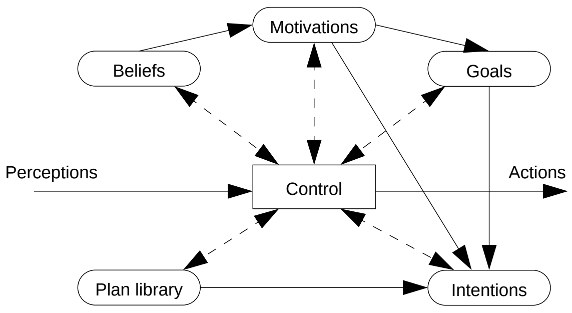


FIGURE 4.1: Griffiths and Luck mBDI architecture.

This model underpins the mBDI architecture [Griffiths and Luck, 2003], which in turn is based on the PRS architecture, plus motivations. Here, motivations are updated by an agent's beliefs, and in turn, influence the adoption of goals and the selection of intentions, as illustrated in Figure 4.1, in which solid arrows represent the flow of control and dashed arrows represent the flow of information.

The reasoning cycle for an mBDI agent starts with an agent perceiving the environment, and using this information to update its belief base. In turn, the now updated belief base is used to calculate the intensity of each motivation, according to the intensity update function f_i. After updating motivational intensity values, an agent compares the intensity of each motivation against its threshold for activation, and if this threshold is exceeded, the goal generation function f_g is invoked to generate new goals. Once new goals are generated, all goals are assessed in relation to the motivation that generated them. The goal with the largest intensity value is slated for achievement, resulting in a plan being selected to achieve it, and it is adopted as an intention. Then, the intention that provides the largest motivational reward is selected from among the set of existing intentions, and a step is executed from it. Finally, when an intention finishes executing, the mitigation function f_m is invoked to reduce the intensity of the associated motivation. These steps are illustrated in Algorithm 2.

The model of motivations used in the mBDI architecture was intended for a procedural agent architecture (*i.e.* the Procedural Reasoning System), and as such it equates the accomplishment of a goal to the complete execution of a plan. This is is apparent from Steps 11 and 13 of the control cycle in Algorithm 2, which describe an intention as a plan to be executed, and mitigation of a motivation as resulting from the completion of that plan. However,

Algorithm 2 mBDI control cycle.

```
1:  loop
2:    perceive the environment and update the beliefs;
3:    for all motivation $m$ do
4:      apply $f_i$ to $m$ to update $i$;
5:    end for
6:    for all motivation $m$ do
7:      if $i > t$ then
8:        apply $f_g$ to $m$ to generate new goals;
9:      end if
10:   end for
11:   select a plan for the most motivated of these new goals and adopt it as an intention;
12:   select the most motivationally valuable intention and perform the next step in its
      plan;
13:   on completion of an intention apply $f_m$ to each motivation to reduce its intensity;
14: end loop
```

the way in which motivations are mitigated results in a sort of *all or nothing* approach to plan execution, since if a plan fails, no mitigation ensues. It is therefore easy to imagine situations in which motivations can be partially mitigated even if the initial plan adopted to satisfy it is not executed fully. For example, if I have a motivation to refuel my car for a short trip, and adopt a plan to fill up its petrol tank, but the fuel in the petrol station runs out after the tank is half full, my initial plan failed, but it is certainly not true that my motivation has not been, at least partially, mitigated.

Furthermore, in the model developed by Griffiths and Luck, the three functions that drive motivated behaviour during agent execution are defined as *abstract* functions. As a result, a number of issues arise regarding their specific operation in a concrete setting. In particular it is not clear:

- how beliefs lead to modifications to motivational intensity;
- how goals are generated when the motivation threshold is exceeded;
- whether each motivation has a goal that is *always* generated, or if goal generation is conditional; and
- how motivation intensity is mitigated.

As a result of these issues and the model's inability to deal with goals to be introduced in Chapter 3, we extend mBDI to address them.

4.4 Extended Model: mdBDI

In this section, we extend the mBDI motivational model in order to address the issues raised above. First, we need to modify the model so that it can apply not only to the notion of goal achievement as plan execution, but also to goal achievement through desired world-states, or goals to be. The idea here, is that motivations must be affected by an agent's perception of world-states, so a particular motivation must only be mitigated when an agent acting to satisfy it perceives that a certain desired world-state holds. This entails that the mitigation function, originally executed as a result of an intention finishing execution, must now be associated with the achievement of particular world-states.

To solve this problem, we must define the basic mechanism behind the mitigation function in terms of world-states. Thus, the mitigation function of our extended model must use a mechanism similar to the original intensity update function, so that it takes an agent's beliefs as its inputs, and evaluates whether or not the currently perceived world-state supports the mitigation of a given motivation. Moreover, support for declarative goals leads to the dissociation of mitigation from intention execution. Mitigation can only occur when a certain motivation is driving behaviour, but since the original mBDI model keeps track of active motivations through the intention it adopted to achieve a motivated goal, by dissociating mitigation with plan execution, we need to introduce the notion of *active* motivations into the resulting reasoning cycle. In our model, an active motivation is a motivation with its intensity level greater than its activation threshold, signalling that a goal to mitigate it must be adopted. When a motivation is active, it can be mitigated, and therefore the mitigation function is used together with the intensity update function to determine motivational intensity.

The resulting reasoning cycle for our mdBDI (motivated-declarative BDI) model, detailed in Algorithm 3 thus contains important differences. The initial perception of the environment and update of motivations occurs in the same way as in the original algorithm, but since we mitigate motivations using the same beliefs used for intensity updates, we apply the mitigation functions immediately afterwards, but only to the *active* motivations. When a motivation is active, its associated goal generation function is invoked, generating new goals, and resulting in it being added to to the list of active motivations. The final part of the algorithm is also similar to the original, consisting of selecting a plan to achieve the most motivated new goal, adopting it as an intention, and executing the next step of the most motivationally valuable intention.

Here, we maintain the original meaning of the intensity update function as a mapping of beliefs into intensity values, and the goal generation function as a mapping from beliefs into new goals. In our extended model, however, the mitigation function is no longer associated with plan execution, but rather in the achievement of world-states, so instead of using the original mBDI f_m function, we define a *declarative* mitigation function f_{md} containing a

Algorithm 3 mdBDI control cycle.

```
1: loop
2:   perceive the environment and update the beliefs;
3:   for all motivation $m$ do
4:     apply $f_i$ to $m$ to update $i$;
5:   end for
6:   for all Active motivation $m \in A_m$ do
7:     apply $f_{md}$ to each active motivation to reduce its intensity;
8:   end for
9:   for all motivation $m$ do
10:    if $i > t$ then
11:      apply $f_g$ to $m$ to generate new goals;
12:      add $m$ to the list of active motivations $A_m$
13:    end if
14:  end for
15:  select a plan for the most motivated of these new goals and adopt it as an intention;
16:  select the most motivationally valuable intention and perform the next step in its
     plan;
17: end loop
```

mapping between *beliefs* and new motivational intensities. This is illustrated in the diagram of Figure 4.2.

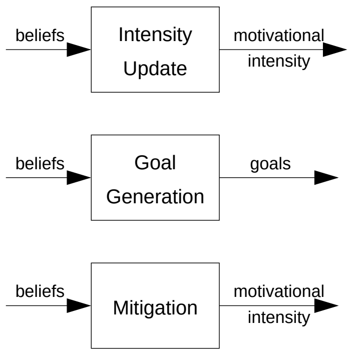


FIGURE 4.2: Inputs and outputs of the motivation functions.

It is important to note that although our modifications are small, they have a number of implications to the way in which motivations generate goals and are mitigated. First, due to the new method of mitigating intentions introduced in our model, this last step of selecting an intention implies the capability of an agent to predict the results of a plan after it is executed, otherwise the agent cannot determine how motivationally rewarding a plan is. This prediction of the effects of plans is further developed in Section 4.6. Second, with this new control cycle, plans may execute successfully and still fail to bring about the desired world-state, but a motivation can only be mitigated by achieving a certain desired world-state. For example, if I want to mitigate a motivation to have a video-game console in my home, I may execute my plan to order one from an internet shop successfully, but if the mail service delays the delivery or loses the package, my motivation has not been mitigated by my plan. Moreover, motivations may be mitigated *regardless* of the plan adopted to do so. Using the same example, if a friend gives me the video-game console as a gift after my initial plan failed, my motivation is still mitigated.

In mBDI, the three functions within each motivation are represented as arbitrary mechanisms that cannot be modified by a designer. But as we have established earlier in the chapter, in order for motivations to be used as a general purpose meta-reasoning abstraction, a designer must be able to specify the details of each motivation function without modifying the associated agent architecture. Therefore, our model must allow custom mappings between an agent's data structures and the outputs of each function.

In the following sections we examine each part of a motivation specification in more detail discussing how the representation of each part of a motivation must be in order to satisfy this requirement, starting with the definition of an individual motivation in Section 4.4.1, and following to each of the three functions in Sections 4.4.2, 4.4.3 and 4.4.4.

4.4.1 Overview of a Motivation

At a high level, each motivation is composed of an identifier Id, an intensity value Int, a threshold T, and the name of a concrete function to be used for each of the required abstract functions of our motivation model, as follows:

$$\langle Id, Int, T, f_i(Beliefs), f_g(Beliefs), f_{md}(Beliefs)\rangle$$

Given the intended meta-level function of our motivational model, we want the computational representation of the motivation to be as simple as possible, limiting the amount of computational resources required in its processing. Thus, we adopt an integer representation of the intensity value of a motivation, eliminating the cost of performing floating point arithmetic during motivational processing. Moreover, since the threshold value must be defined in relation to the intensity value, it must also be represented as an integer. The threshold value can be any integer value, so that a designer may define any scale for the intensities in a motivation, and it is perfectly possible to have a threshold value of 100 and define the evolution of the intensity update function in increments of 10.

We have seen that whenever a motivation becomes active, the goal generation function is invoked, after which the mitigation function is invoked to verify if the condition for the motivation to be mitigated is reached. The output of each motivation function must be defined by a mapping process compatible with the purpose of the function, so if we are dealing with the intensity update or mitigation functions, the mapping consists of belief-value correspondences, while if we are dealing with a goal generation function, the mapping is a series of belief-goal associations, since this function aims to generate goals, given the perceptions that activated a motivation.

4.4.2 Intensity Update Function

The first function in our model is the intensity update function, which is responsible for translating belief and perceptual data into new values that represent changes in the intensity of the motivation associated with this function. As a consequence, the definition of an intensity update function must consist of a mapping between logical expressions over an agent's belief base (following the AgentSpeak model) into integer arithmetic expressions (following the scale of the intensity value) defining the motivational value of the preceding logical expression. By providing a mapping mechanism, we depart from the arbitrary mechanism present in the original mBDI [Griffiths and Luck, 2003] model into one in which a designer can specify any correspondence between belief expressions and intensity values.

For example, consider the postman robot of Section 3.2.2. Here, a designer may wish to create a motivation for processing packets arriving from a particular loading bay, and define a function that causes the intensity of this motivation to increase by one unit whenever a packet sits on top of this loading bay. In this situation, the designer must create a rule stating that if the motivated agent believes there is a packet over the specified loading bay (*i.e.* over(P,bay1)), then the motivational intensity is changed by one unit.

Mapping rules can be more complex than that, as a designer may need to define a composition of conditions that results in slower motivation accumulation depending on certain conditions. For example, if the designer of the same postman robot now wants this motivation to only increase in intensity if the robot's battery is at level 10, and if the robot is occupied the rate of motivational accumulation should be diminished by half, multiple rules must be defined. The resulting intensity update function can be abstractly represented as follows:

$$f_i(Beliefs) = \begin{cases} over(P, bay1) \wedge batt(10) \rightarrow 2 \\ occupied(robot) \rightarrow -1 \end{cases}$$

Here, the intensity of the motivation to process a packet is increased by 2 whenever the agent believes a new packet has arrived in loading bay 1 (*i.e.* bay1) and it has a battery level of 10. It is important to notice that this language deals exclusively with beliefs, both intrinsic ones and those resulting from perception, whereas some motivation models assign values to actions and, by doing so, conform to a procedural view of reasoning.

4.4.3 Goal Generation Function

The second function underpinning our model of motivations is the goal generation function which, as the name implies, is a function that causes an agent to adopt goals. We have seen earlier that our language must be conceptually compatible with AgentSpeak(L), so the goal representation used in a goal generation function specification must be that of the AgentSpeak(L) language.

One simplistic way of defining this function is to specify a static set of goals that are posted to the agent whenever the associated intention's intensity surpasses its threshold. Nevertheless, we define a goal generation function based on a mapping from belief expressions (like in the intensity update function) into goals. For instance, taking the postman robot of the previous example, we can create a function which, whenever the threshold is exceeded, generates the goal to sort the packet situated at loading bay 1 (*i.e.* bay1), if the robot is not occupied, as follows:

$$f_g(Beliefs) = \Big\{ over(Packet, bay1) \wedge notoccupied(robot) \rightarrow +!sort(Packet)$$

There are two reasons for this particular type of mapping. The first is that the simplistic approach ignores the possibility that an agent may not be ready to act immediately after the threshold is exceeded. In our example, we only want the robot to start sorting the packet if the robot is not occupied. This restriction allows a designer to prevent goals from being posted that will interfere with other possibly conflicting goals, keeping the motivation unmitigated so that when the robot is no longer occupied it can immediately adopt this goal. This representation also prevents goals from being adopted in situations in which they would fail. The second reason for this representation is that the belief expression part of the rule allows goals to be *bound* to variable objects in the belief base. In the example, the identity of the specific packet that must be sorted cannot be known at design time, so the belief over(Packet,bay1) allows the function to map the variable Packet to be bound to whatever packet the robot believes to be over bay1 at runtime and adopt a goal accordingly.

4.4.4 Mitigation Function

The last function we consider in the model is the mitigation function, which is invoked after a motivation has been activated to mitigate the motivation. As we have seen, our extension to the mBDI motivational model aims to accommodate the notion of goals to be, as opposed to the purely procedural view taken by the original model, in which the mitigation function is always invoked whenever a plan is executed to mitigate an intention adopted through motivation. In order to accomplish this, our new mitigation function must be able to identify the desired world-states that the motivated goal is intended to achieve, and only then mitigate the intention. Consequently, the resulting function must include some mapping of beliefs (representing the desired world-state) into some specified intensity value that represents the mitigation of the associated motivation.

Following our running example of the motivation to sort packets arriving from a loading bay by a postman robot, we identify the world-state intended for this motivation to be mitigated as the one in which the packet is over the pigeonholes in the mail warehouse. When this state is reached, we mitigate the motivation by subtracting 20 units from the motivational intensity. This can be represented in the following function:

$$f_m(Beliefs) = \Big\{ over(Packet, pigeonHoles) \rightarrow -20$$

Therefore, from a purely representational perspective, the mitigation function is exactly the same as the intensity update function in that it consists of a mapping between beliefs into intensity values. The difference lies in the way in which the mitigation function is used during the reasoning cycle, so that, after a motivation is activated, the mitigation function identifies the particular desired world-state that denotes that the motivation should be mitigated.

4.5 A Language of Motivation

The second major issue we need to address is the absence of a concrete specification for the three functions used in the model. We have seen that the mBDI model provides the abstract machinery that drives motivated control, but it is necessary to associate these abstractions to concrete instances of motivations. As we stated in the requirements we set out in Section 4.1, in order to bind the variable aspects of a motivation to specific functions (*e.g.* the association of motivation updates to specific world-states) in a generic and reusable way, we need a specification language bound to our model of motivations. Therefore, in this section, we provide bindings between the specification language and the abstract requirements of the three functions mentioned in Sections 4.3 and 4.4 that drive motivated behaviour. We start this section with the general requirements for the language, and then consider the requirements of the individual functions.

4.5.1 Requirements

The main requirement of our language relates to the individual aspect of each agent's motivations. Since different individuals can have different sets of motivations, these individuals are affected by their motivations in varying ways, each with its own dynamics to allow evaluation of situations and achievement of goals according to an agent's unique priorities. So, the way in which we set out to use motivations to describe meta-reasoning aims to allow a designer to describe motivational aspects for each agent. Thus we require a language to describe unique sets of motivations based on the abstract functions and data structures of the mdBDI model. In consequence, we have designed a language centred on the three abstract functions: intensity update; goal generation; and mitigation. Concrete versions of these functions are essentially mappings between beliefs and an intensity value in the case of intensity update and mitigation, or new goals for the goal generation function. These functions are specified for each individual motivation, of which an agent can have several.

Furthermore, the agent language considered in this thesis is AgentSpeak(L), which leads to the second requirement of our language of motivations, which is conceptual compatibility with AgentSpeak(L). In order to accomplish this, we must use the same language elements from AgentSpeak to represent beliefs, goals and plans. We show how these elements are represented in a concrete language implementation in Section 4.5.2, exemplifying this language in Section 4.5.3.

4.5.2 Language

Having seen how the requirements of Section 4.5.1 are translated into the conceptual framework of our motivation language, we now proceed to introduce the syntax of the language. As stated in Section 4.4.1, the high-level structure of a motivation contains the motivation identifier, its intensity, its activation threshold and the three functions of intensity update, goal generation and mitigation. From a specification perspective, a motivation's initial intensity does not need to be directly specified, as it can be assumed to start with a null value, so our syntactic representation does not include it. In our language, the other language elements are specified using an organisation similar to a class definition in an object-oriented language. These basic elements of a single motivation are shown in the excerpt of Table 4.1, and follow the example of the postman robot's motivation to process packets arriving at a loading bay. In this excerpt we can see that a motivation's name is stated after the keyword Motivation, that the threshold value is stated after the keyword Threshold, and that motivation functions are specified immediately afterwards.

```
Motivation processBay {
    Threshold = 10;

    IntensityUpdate MyIntensityUpdateFunction { ... }
    GoalGeneration  MyGoalGenerationFunction { ... }
    Mitigation      MyMitigationFunction { ... }
}
```

Table 4.1: High-level description of a motivation.

Each motivation function is preceded by a keyword indicating which type of function is being defined followed by the name of a function implementation, so that the *intensity update function* is preceded by the IntensityUpdate keyword, the *goal generation function* is preceded by the GoalGeneration keyword, and the *mitigation function* is preceded by the Mitigation keyword. It is important to note that the specification of a *function implementation* following the function type keyword implies that in our implemented system it is possible to change the exact way in which each function is processed by the motivation model. In this thesis, however, we only consider function implementations that process the mappings described in

Sections 4.4.2, 4.4.3 and 4.4.4. We consider these functions in detail later in this section, but before we proceed, it is important to understand the basic building blocks of the language.

Given the requirement of AgentSpeak(L) compatibility, the elements used in the mappings specified within each motivation function must comply with the type of representation used in an AgentSpeak(L) agent specification. To accomplish this, the grammar of our language includes rules to parse AgentSpeak(L) triggers, literals, atoms, variables, logical expressions and arithmetic expressions in the exact same way as an AgentSpeak(L) interpreter would parse them. This can be seen in the whole BNF of the language, described in this section and shown in Table 4.2, and in which the logical framework has been derived from the BNF of the parser used with the Jason [Bordini et al., 2005b] AgentSpeak interpreter. It is important to note that, like AgentSpeak, we follow the Prolog [Nilsson and Maluszynski., 1995] convention of variables being expressed as literals with a capitalised first letter. Moreover, the unification used in the evaluation of one function carries on to the next one in the update cycle of the same motivation, so for example, if a certain variable `X` in the intensity update function of motivation `M` is unified to a certain value `a`, all instances of `X` in the following functions evaluated for `M` will also be unified to `a`.

As we have seen, the functions for updating the intensity of, and mitigating, a motivation need to provide some kind of mapping between perceptual data and an intensity variation. As a result, our language of motivation allows the specification of a mapping between beliefs and an arithmetic expression indicating how the intensity level should be modified as a result of the beliefs being true. Any specific mapping is represented as:

$$log\_expr- > arithm\_expr$$

where log_expr is a logical expression on the beliefs (*e.g.* a(X) & b(Y)), and $arithm\_expr$ is an arithmetic expression (*e.g.* X+2), as shown in Table 4.3.

An example of such a mapping is shown in Table 4.4, replicating the postman example used in Section 4.4.2. Here, the intensity of the motivation to process a packet is increased by 2 whenever the agent believes a new packet has arrived in loading bay 1 (`bay1`) and has a battery level of 10. It is important to notice that this language deals exclusively with beliefs, both intrinsic ones and those resulting from perception, whereas some motivation models assign values to actions and by doing so conform to a procedural view of reasoning.

The mitigation function provides a mapping that is syntactically the same as the intensity update function, though the function serves a different purpose, as outlined in Section 4.4.4, being only invoked when a motivation is active, to determine when its associated motivation has been mitigated. For example, if the motivation of Table 4.4 is mitigated when a packet `P` has been sorted, this can be expressed as the function of Table 4.5.

Aside from mapping beliefs into perceptions, we must also describe the mapping of beliefs into goals. We have seen in Section 4.4.3 that goal generation is invoked when the motivation

$$
\begin{aligned}
parse &::= (motivation)+\\
motivation &::=< MOTIVATION > identifier``\{"motivationBody``\}"\\
motivationBody &::= threshold``;"intensityUpdategoalGenerationmitigation\\
threshold &::=< THRESHOLD > ``="\ < NUMBER >\\
identifier &::=< ATOM >\\
&\mid < VAR >\\
classname &::= identifier\\
intensityUpdate &::=< INTENSITY\_UPDATE > classname``\{"\\
&(beliefToIntegerMapping``;")*``\}"\\
beliefToIntegerMapping &::= (log\_expr``->"arithm\_expr)\\
goalGeneration &::=< GOAL\_GENERATION > classname``\{"\\
&(beliefToTriggerMapping``;")*``\}"\\
beliefToTriggerMapping &::= (log\_expr``->"trigger)\\
mitigation &::=< MITIGATION > classname``\{"\\
&(beliefToIntegerMapping";")*``\}"\\
trigger &::= (``+"|``-")((``!"|``?"))?(literal|var)\\
literal &::= (((< TK\_NEG >)?atom)|< TK\_TRUE >|< TK\_FALSE >)\\
atom &::=< ATOM > (``("terms``)")?(list)?\\
terms &::= term(``,"term)*\\
term &::= (literal|list|arithm_expr|string)\\
list &::= ``["(term(``,"term)*\\
&(``|"(< VAR >|< UNNAMEDVAR >|list))?)?``]"\\
log\_expr &::= log\_expr\_trm(``|"log\_expr)?\\
log\_expr\_trm &::= log\_expr\_factor(``\&"log\_expr\_trm)?\\
log\_expr\_factor &::= (< TK\_NOT > log\_expr\_factor|rel\_expr)\\
rel\_expr &::= (arithm\_expr|literal|string)\\
&((``<"|``<="|``>"|``>="|``=="|``\backslash\backslash=="|``="|``=..")\\
&(arithm\_expr|literal|string|list))?\\
arithm\_expr &::= arithm\_expr\_trm((``+"|``-")arithm\_expr)?\\
arithm\_expr\_trm &::= arithm\_expr\_factor(\\
&(``*"|``/"|< TK\_INTDIV >|< TK\_INTMOD >)arithm\_expr\_trm)?\\
arithm\_expr\_factor &::= arithm\_expr\_simple((``**")arithm\_expr\_factor)?\\
arithm\_expr\_simple &::= (< NUMBER >|``-"arithm\_expr\_simple|``("log_expr``)"|var)\\
var &::= (< VAR >|< UNNAMEDVAR >)(list)?\\
string &::=< STRING >
\end{aligned}
$$

Table 4.2: BNF of the motivation language.

$$
\begin{aligned}
intensityUpdate &::=< INTENSITY\_UPDATE > classname``\{" \\
&(beliefToIntegerMapping``;") * ``\}" \\
beliefToIntegerMapping &::= (log\_expr``- > "arithm\_expr) \\
mitigation &::=< MITIGATION > classname``\{" \\
&(beliefToIntegerMapping";") * ``\}"
\end{aligned}
$$

Table 4.3: BNF of intensity update and mitigation functions.

```
Motivation processBay {
   ...

   IntensityUpdate MyIntensityUpdateFunction {
      over(P,bay1) & batt(10) -> 2; //This increases intensity
      occupied(robot) -> -1;        //This lowers it a bit
   }

   ...
}
```

Table 4.4: Example of an intensity update function.

```
Motivation processBay {
   ...

   Mitigation MyMitigationFunction {
      sorted(P) -> -20; //This mitigates a motivation
   }

   ...
}
```

Table 4.5: Example of a mitigation function.

threshold is exceeded as a result of intensity accumulation. In turn, our language allows the specification of additional constraints before a goal is generated, or the unconditional generation of goals through the *true* condition, as shown in the BNF of Table 4.6. This mapping is similar to intensity update in that mappings start from a logical expression over beliefs. However, the targets of this mapping are goal addition events. As a consequence, new goals to be achieved are added as a result of the intensity reaching the threshold in the motivation containing this goal generation function.

An example of goal generation function is illustrated in Table 4.7, where the agent generates an event to sort a packet located over $bay1$ whenever the goal generation function is invoked. Here, the constraint on the left of the mapping exists only to allow the unification of the packet name with the goal to be generated.

$$\begin{aligned} goalGeneration &::= < GOAL\_GENERATION > classname``\{" \\ &\quad (beliefToTriggerMapping``;") * ``\}" \\ beliefToTriggerMapping &::= (log\_expr``->"trigger) \end{aligned}$$

Table 4.6: BNF of the goal generation function.

```
Motivation processBay1 {
   ...

   GoalGeneration MyGoalGenerationFunction {
      over(Packet,bay1) & not occupied(robot) -> +!sort(Packet);
      //true -> +!sort(packet1); //Another possibility
   }

   ...
}
```

Table 4.7: Example of a goal generation function.

4.5.3 Language Example

To exemplify how our language can be used to model behaviour, we work through the example of the postman robot. The idea here, is that the example of Section 3.3.3 is modified so that the goal of sorting packets is now driven by motivations rather than as a direct reaction to the arrival of new packets. We introduce a motivation to process packets arriving from loading bay 1 using our language and shown in Table 4.8, associated with an agent almost identical to that of Table 3.8, with the exception that it does not react immediately to events of the type over(Packet,Bay). The intuition for this motivation is that an agent now waits for a certain time before picking up a package and moving it around, allowing for other activities to be carried out before delivering packages. Moreover, when an agent is already occupied in delivering a packet, the motivation to deliver another packet is expected to remain the same, preventing multiple goals to deliver packets from being adopted simultaneously.

In more detail, this example describes the dynamics of a motivation to process packets arriving from loading bay 1. The motivational intensity starts to increase as soon as the agent detects an undelivered packet over bay 1, until it reaches the threshold of 10. Once the threshold is reached, the goal generation function adds a goal to deliver this packet. Finally, the agent assumes the motivation is mitigated when it perceives the packet to be over the pigeonholes, diminishing the motivational intensity accordingly. Here, if an agent needs to monitor multiple bays at the same time, it will give priority to the one that has the highest motivation.

```
Motivation processBay1 {
   Threshold = 10;                    //This is the threshold
   IntensityUpdate org.kcl.nestor.mot.impl.
      IntensityUpdateFunctionImpl {
         over(P,bay1) & batt(10) -> 2; //This increases
            intensity
      occupied(agent) -> -1;         //This lowers it
   }

   GoalGeneration org.kcl.nestor.mot.impl.
      GoalGenerationFunctionImpl {
      over(Packet,bay1) -> +!deliver(Packet);
   }

   Mitigation org.kcl.nestor.mot.impl.MitigationFunctionImpl {
      over(Packet, pigeonHoles) -> -20;
   }
}
```

Table 4.8: Example of a set of motivations.

4.6 AgentSpeak(MPL): A Motivated AgentSpeak Interpreter

Autonomous agents are expected to generate goals pro-actively instead of simply reacting to discrete events in the environment [Duff et al., 2006]. Generating goals pro-actively entails that an agent has a way of assessing its current situation and anticipating how the environment (or other agents in the environment) will behave, in order to provide a rational justification for the adoption of a goal. Since motivations can be used to associate a measure of importance to goals, it is possible to use motivational intensity to guide an agent's choice of action when faced with multiple conflicting courses of action.

In traditional AgentSpeak, plans are adopted as a reaction to events in the environment in a direct sense. That is, plans are expressed so that if a certain event e happens in a certain world-state, an agent having a plan with a matching triggering event e always adopts this plan. Furthermore, since goals in the procedural sense used by AgentSpeak(L) are adopted as part of the execution of plans, an agent does not generate them through deliberation, and they are instead adopted in the process of reacting to some event in the environment. For instance, a plan may be described so that whenever an agent believes that a given block is on a table (*e.g.* on(block,table)), a procedure to remove such a block is invoked. This amounts to simple reaction rather than deliberate, future-directed behaviour. This method of *behaviour selection* also fails to properly describe the reasons for goal adoption in a declarative sense. Such shortcomings can be exemplified using the traditional example of a block on a table, in which a declarative goal to remove the block from the table could be described as **not** on(block,table). The question here is whether an agent should *always* react to new events and start deliberation immediately even if this agent might be pursuing other, more important, goals. Meta-level control is intended to address this issue by allowing reasoning to take place before action-directed plans are adopted, and our motivation model provides the means with which to compare goals through their relative worth to an agent.

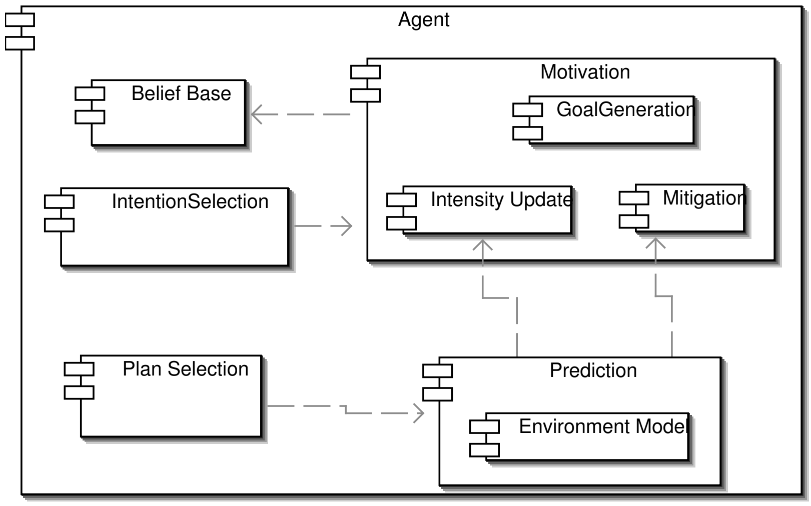


FIGURE 4.3: Modules of the motivated AgentSpeak.

AgentSpeak has three key reasoning processes that must be modified in order to accommodate the meta-reasoning model we defined in Section 4.2: belief update, plan selection and intention selection. In the following sections, we show how we modify the reasoning processes of AgentSpeak processes to take the motivations model into account. The resulting architecture is AgentSpeak(MPL), an extension to our own AgentSpeak(PL) architecture that contains a motivation module that drives an agent's goal adoption and plan selection process. The components of this architecture are summarised in Figure 4.3, which shows the motivation module including its three functions and their association with the belief base and intention selection processes, explained in Sections 4.6.1 and 4.6.3, as well as the prediction module associated with plan selection, explained in Section 4.6.2. We close the section with an example summarising how the motivation module affects the adoption and prioritisation of intentions in Section 4.6.5.

4.6.1 Motivations and Belief Update

In our model, motivation intensity is a function of the perceived world-state, so most of the motivation machinery is associated with the agent's belief update function. Each motivation data structure comprises an intensity value, a threshold value, and functions for intensity update, goal generation and mitigation. These data structures are updated as a result of the agent perception of the world, as illustrated in the activity diagram of Figure 4.4. When an agent receives new perceptions, it updates its belief base which is immediately inspected by the *intensity update function* associated with all motivations affecting this agent. During the update process, if the motivational intensity of any motivation reaches its threshold level, the *goal generation function* is invoked, also inspecting the belief base, and generating a set of goals based on the current world-state. Finally, the belief base is inspected by the mitigation function to determine if any of the motivations triggered previously have been mitigated, in which case motivations are adjusted accordingly.

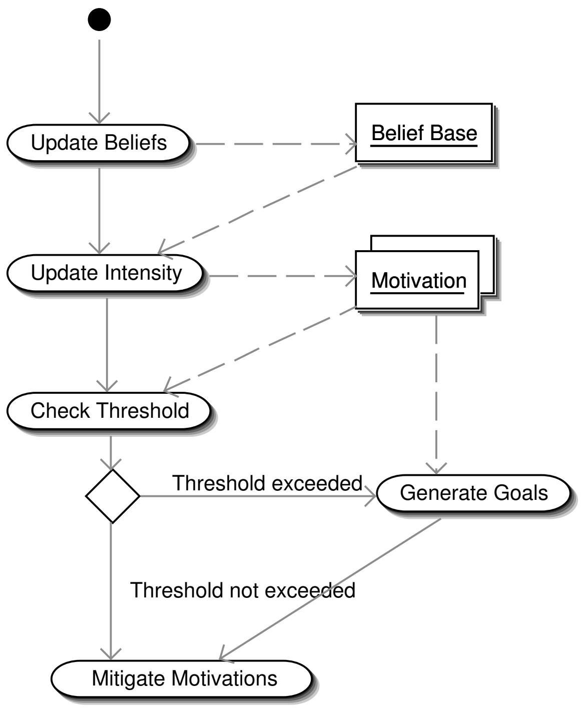


FIGURE 4.4: Activity diagram of a motivated belief update.

In practice, the intensity update performed by the mitigation function is equivalent to that of the intensity update function. However, this update can only apply to motivations that had been previously *activated* as a result of their threshold levels being reached, and had generated goals.

4.6.2 Motivations and Plan Selection

The second modified process in AgentSpeak(L) is plan selection. Since goals are adopted based on a numerically quantified (motivational) importance, it makes sense to use this quantification as a criterion for plan selection. Information regarding the motivation that led an agent to adopt a certain goal can be used in the selection of the best course of action to mitigate this motivation. Since our model provides separate functions for motivation intensity update and mitigation, it is possible to use them along with a world-state prediction model to determine the motivational reward of executing the plans known by the agent. Therefore, we need a motivation-driven plan selection function, which selects the most motivationally rewarding plan from a set of applicable plans.

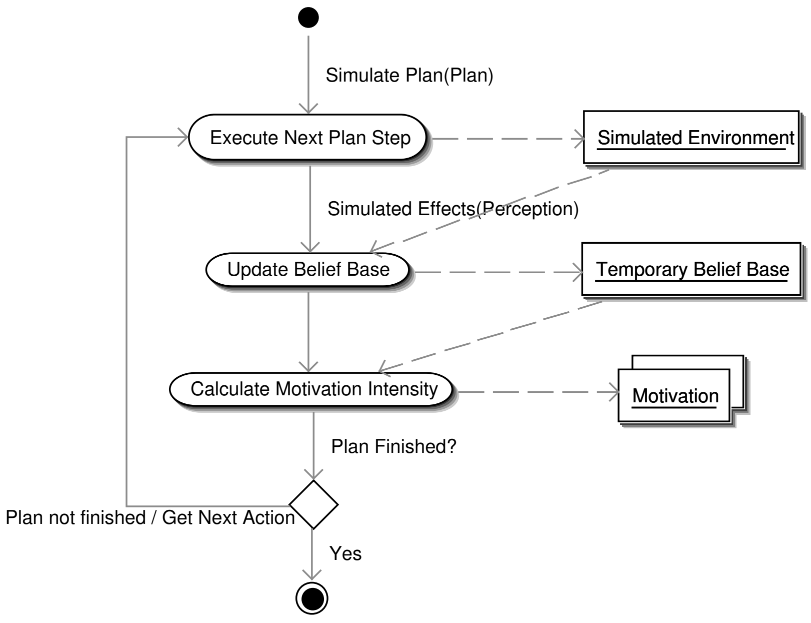


Figure 4.5: Activity diagram of the prediction module.

In order to calculate this quantification, plans are submitted to a prediction module that consists of a *sandbox*[2] [Gong et al., 1997] copy of the current belief base, as well as a simplified model of the environment. This prediction module then simulates the execution of the plan using the functions from the motivation module to determine the expected overall decrease in motivational intensity. The diagram of Figure 4.5 illustrates the operation of prediction module as consisting of executing each step in a plan while determining its effects over the simulated beliefs, followed by the application of the motivation update function over these simulated beliefs to establish the motivational reward of the plan. Once all predictions are collected, the function selects the plan that provides the largest mitigation reward. Alternatively, a designer may specify the declarative outcome of each plan and use this outcome to calculate the expected motivational reward.

4.6.3 Motivations and Intention Selection

When an agent has committed itself to achieving multiple concurrent goals, their relative priorities may fluctuate as events take place in the environment while the agent carries out plans to achieve these goals. Achieving goals in a timely fashion is crucial in a highly dynamic environment, and an agent has to adapt to changing circumstances with minimum overhead. As we have seen, an AgentSpeak interpreter creates a new intention for every *external* goal adopted by an agent (*i.e.* not subgoals), which contains the plan to achieve

[2]In computer security and software development, a sandbox is an insulated environment where tests can be performed without affecting the main system.

this goal, as well as the plans required to achieve any possible subgoal generated in the process of carrying out the initial plan.

Subsequently, the interpreter selects an intention to execute the steps from the plans included in them. In a traditional AgentSpeak interpreter, this selection works on a first-come first-serve basis. In our motivated agent architecture, external goals are generated by the *goal generation function* whenever a threshold intensity is reached in a motivation. Here, using motivation information to prioritise goals can help an agent achieve important goals quickly by postponing the execution of plans to achieve less important goals. Under this assumption, we require a module that evaluates all active intentions and selects the intention associated with the most motivated goal. However, one potential problem with this approach to goal prioritisation is that it can lead to problems in the case of plans that gradually mitigate a motivation as they are executed. In this scenario, this kind of plan will be constantly preempted by new plans adopted to mitigate higher intensity motivations, possibly leading to it never being finished. This behaviour might be undesirable in certain situations, so it is important to ensure good motivation design to avoid this effect in these situations.

4.6.4 Motivation update and timing

The mBDI architecture does not specify the behaviour of the goal generation function between threshold activation and goal mitigation. The effects of this ambiguity become apparent when there is a significant delay between goal adoption and goal achievement. For example, consider a *nourishment* motivation that generates a goal to feed whenever its threshold is reached, and for which all available plans take three units of time to be executed before the goal is achieved. Moreover, suppose that the agent performs one reasoning cycle per unit of time so, assuming the plan is successful, the agent will perform three reasoning cycles before mitigating this motivation. In the meantime, it is not clear whether or not the agent should generate the same goal three times until the motivation is mitigated or generate goals only once between a motivation's threshold being reached and its subsequent mitigation.

In our implementation, we take the approach of generating a *single* goal when a threshold is exceeded rather than multiple goals as time elapses. However, it might be interesting to add a language construct to determine when a motivation can generate multiple goals per activation.

4.6.5 Example of motivation dynamics

In order to better illustrate the mechanism described in this section, we now develop an example of how a set of motivations affects the reasoning cycle of an AgentSpeak(L) agent.

Motivation	Threshold	Goal
motivationA	15	*goalA*
motivationB	10	*goalB*
motivationC	20	*goalC*

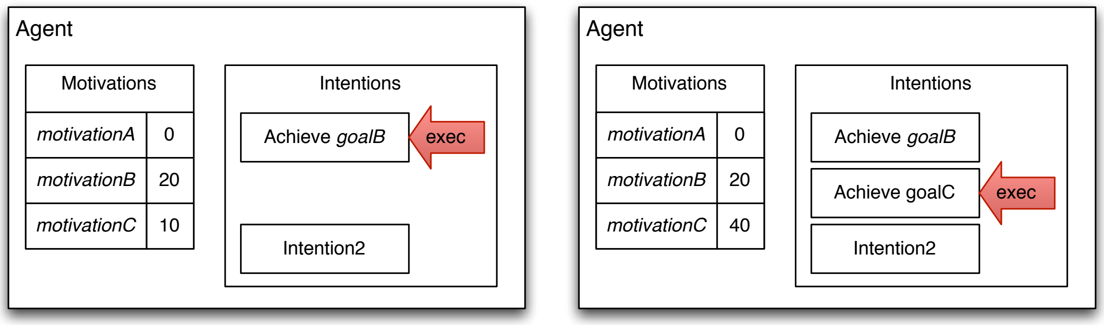
TABLE 4.9: Example motivations and activation thresholds

We consider an agent containing three motivations, *motivationA*, *motivationB* and *motivationC*, with associated goals and activation thresholds as summarised in Table 4.9. Since the specific manner of motivation accumulation and mitigation is irrelevant for their impact on the reasoning process, we ignore the intensity update function and the mitigation function for this example. Each of these motivations has a fairly simple goal generation function that generates a single goal whenever a certain motivation is activated.

If, during the execution of this agent, the intensity of *motivationB* reaches a value of 20, the motivation becomes active, leading to the adoption of a plan to achieve *goalB*. Assuming that no other motivation is active, the intention corresponding to the plan to achieve *goalB* will be given priority to be executed by the intention selection function, as illustrated in Figure 4.6(a). Now, if while executing the intention to achieve *goalB* the environment changes in such a way that the intensity of *motivationC* reaches its threshold of 20, a new intention to achieve *goalC* (*i.e.* a plan to achieve *goalC*) is also adopted. Moreover, if the intensity of *motivationC* surpasses the intensity of *motivationB*, then this new intention is given priority over all other motivations, as illustrated in Figure 4.6(b). It is important to notice in this example that, since *motivationA* was never activated, no intention to achieve its associated goal was ever adopted.

4.7 Experimental Evaluation

In the evaluation of our AgentSpeak(L)-based motivated architecture, we developed a practical experiment in which the motivation language is used to augment an AgentSpeak(L)



(a) Execution of plan to achieve *goalB*. (b) Priority given to intention to achieve *goalC*.

FIGURE 4.6: Effect of motivations on intention selection.

agent specification. The experiment was adapted from an existing scenario of a mars rover and aims to provide a basis for comparison between a meta-reasoning enhanced agent and a more traditional approach. We describe the scenario in Section 4.7.1 and show the empirical results of our evaluation in Section 4.7.2.

4.7.1 Mars Rover

In order to evaluate the potential improvements to agent efficiency, we adapted the scenario used by Duff *et al.* [Duff et al., 2006] to outline the advantage of proactive maintenance goals in agents. This scenario consists of a Mars rover capable of moving about a two-dimensional environment, in which movement consumes energy from the rover's batteries as it moves. Each unit of distance covered by the rover drains one unit of battery energy, and the rover can recharge at a mothership located at the centre of the environment. In the scenario, goals consist of waypoints through which the rover must pass in its exploratory expedition. A varying number of goals was given to the agent to assess four test parameters:

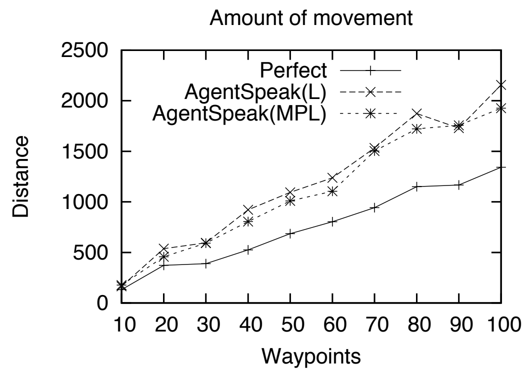
- effective movement, consisting of the distance travelled towards waypoints;
- supply movement, consisting of the distance travelled towards the mothership for recharging;
- wasted movement, consisting of the distance travelled to a waypoint that was wasted due to the agent needing to recharge halfway through getting to a waypoint; and
- the number of intentions dropped to avoid complete battery discharge.

In this context, a more effective agent travels the least wasted distance, as well as the least supply distance, thus optimising battery use[3]. Regarding the reasoning process itself, a more rational agent can be seen as one that drops the least amount of goals, since reasoning about adopting and managing ultimately dropped goals is also wasteful.

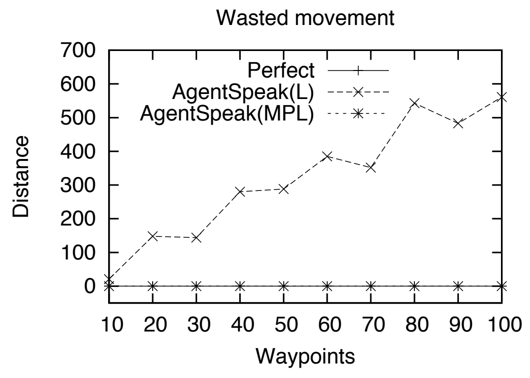
Our experiments consisted of submitting an increasingly larger number of waypoints, randomly generated for each set, varying from 10 to 100 waypoints, to three different agents. The baseline of performance for the experiments was established by an agent with an infinite amount of energy, giving optimal movement to navigating through the entire set of waypoints, since there is no need to move to the mothership.

These waypoints were also submitted to a traditional AgentSpeak(L) agent, which must monitor its battery level and recharge before being too far from the mothership. Its strategy is to navigate to the waypoints in the order they arrive, without prior consideration of battery level. Whenever it is about to move to a position beyond reach of the mothership,

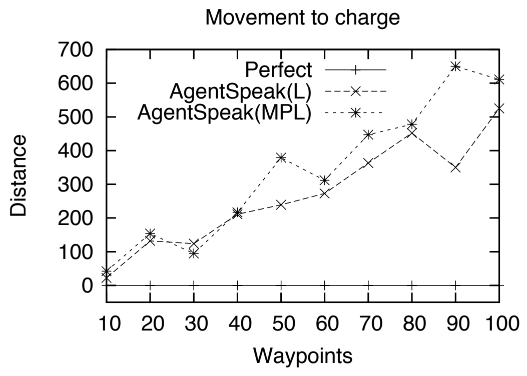
[3]The Pathfinder [Mishkin et al., 1998] rover mission ceased its operations due to the limited number of recharges of its onboard battery.



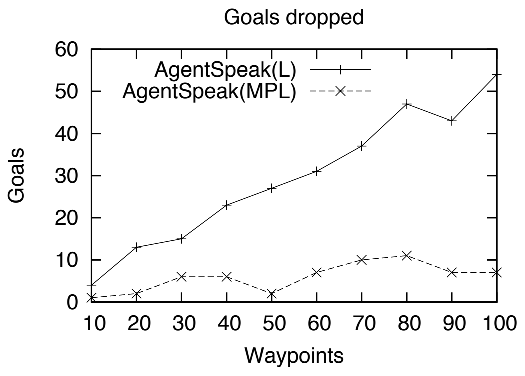
(a) Total distance covered.



(b) Wasted movement.



(c) Movement towards the mothership.



(d) Number of dropped goals.

FIGURE 4.7: Graphics for the rover experiment

it drops the goal to navigate the waypoints and adopts the goal to move to the mothership and recharge.

A third agent used AgentSpeak(MPL), driven by two motivations to navigate and to keep a safe battery level. These motivations behave as a sort of bilateral inhibition mechanism, in which a high intensity for the motivation to have a safe battery level suppresses the intensity of the motivation to navigate. In more detail, whenever it perceives a new waypoint, the motivation to navigate is stimulated, and when it reaches its threshold, a goal to navigate to this waypoint is generated. The motivation to navigate, however, is suppressed if the battery level is not sufficient to reach that waypoint or if the charge spent doing so will place the agent in a position too far from the mothership. Conversely, the motivation to keep the battery level safe is stimulated by these two battery-related conditions. When the intensity of this motivation reaches its threshold, a goal to move to the mothership and recharge is generated. The result of this is that a goal to navigate to a waypoint should not be generated unless the rover has enough battery to move to it and then to the mothership.

4.7.2 Results

Because the traditional AgentSpeak(L) agent starts executing plans to navigate as it perceives new waypoints, it only detects a critical battery level after having moved some distance towards its target position, resulting in a waste of movement actions and a larger total distance covered. On the other hand, the effect of considering the amount of charge before starting to execute a plan to navigate is that no movement is wasted, and thus the total distance covered is smaller, as illustrated in Figure 4.7(a), which compares the total distance covered by each of the agents. In these experiments, the motivated agent had to cover an average 6% less distance than the traditional AgentSpeak(L) agent. The traditional agent had to cover an average of 54% more distance than the baseline agent, compared to 45% for the motivated one, as illustrated in Figure 4.7(b).

Figure 4.7(c) illustrates the distance covered while moving towards the mothership for charging, where the motivated agent appears to move more than the traditional agent. This is a side-effect of the motivated agent always moving towards the mothership *intentionally*, rather than a side-effect of moving towards a waypoint closer to the mothership, which is corroborated by the smaller amount of total movement shown previously.

The last evaluation parameter for this experiment relates to the number of goals dropped as a result of higher-priority goals being pursued. Goals to navigate must be dropped by the rover whenever it has to move back to the mothership to recharge. Goals that are eventually dropped amount to wasteful deliberation, which rational agents should minimise. Here, the difference between the two approaches is more pronounced, with the motivated agent dropping an average of 75% fewer goals than the traditional agent, as shown in Figure 4.7(d).

4.7.2.1 Specification size

In terms of the size of the agent specification, illustrated in Table 4.10, traditional AgentSpeak(L) uses a larger plan library to perform the meta-reasoning required to manage concurrent goals, and to allow the prioritisation of the goal to recharge. On the other hand, the motivated agent's plan library, which was derived from the former, requires a significantly smaller number of plans, since the motivation module ensures that goals are generated once per motivation until being mitigated, while also taking care of bilateral coordination.

	AgentSpeak(L)	**AgentSpeak(MPL)**
# atoms	119	95
# plans	19	10

Table 4.10: Plan library size comparison.

4.8 Conclusion and Discussion

In this chapter we have described an extended agent architecture that includes an explicit meta-reasoning module with an accompanying motivation-based specification language. This allows the specification of rules for adopting goals, selecting plans and prioritising currently adopted intentions to satisfy some subjective definition of importance, defined in terms of an agent's motivations. Motivations are specified separately from the agent language, allowing for both the reactive plan adoption mechanism inherent to traditional agent languages and the proactive type of goal adoption enabled by motivated reasoning.

While our mechanism is not as precise as some other attempts to quantify rewards for certain behaviours, it is a simple mechanism that involves minimal overhead in the interpreter reasoning cycle. Our experiments have shown that our model can achieve the same kind of improvement that other reasoning optimisation strategies have achieved. In addition, specifying meta-level behaviour separately from action-directed behaviour also results in a simpler agent description. Finally, although the motivation mechanism described in Section 4.2 is a modification of previous work on motivations associated with a PRS-like interpreter, we believe our model is generic enough that it can be used in most current agent interpreters, such as 3APL or JACK.

Chapter 5

Social Agentspeak(L)

5.1 Introduction

Although Chapter 3 describes the applicability of planning algorithms in the generation of new individual plans through composition of existing plans in a plan library [Meneguzzi and Luck, 2007a], allowing an agent to discover new ways of achieving its goals by combining its existing plans into new high-level plans, agents are still limited to their individual capabilities. When operating in a society, however, agents can resort to cooperating with others to overcome their limitations. In the context of a cooperative multiagent society with a highly dynamic set of available capabilities, the ability to combine newly discovered capabilities to achieve goals is an important feature, yet the generation of new plans based on previously unknown third party capabilities has only received little attention.

More specifically, when a planning-capable agent needs to achieve a new goal, it searches its plan library for applicable plans. When no suitable plan is found in the plan library, the planner is invoked in an attempt to generate a new plan to satisfy the desired goal. Plans generated by the planner in this way are added to the plan library and become available to the agent to solve future instances of that particular goal. However, if the planner fails to generate a new plan, this (normally) means that the proposed goal is impossible to achieve, given the world-state and capabilities known to the agent at the time of planning. Such an approach to planning has been studied for individual agents, planning over their own capabilities, under the premise that an agent's capabilities are static throughout its life cycle.

However, in a multiagent environment the limitations of an individual agent may be overcome with the help of others, either by delegating tasks to other agents, or by learning new ways of achieving goals. This means that the assumption that the set of available capabilities is static no longer holds. In this case, a failure to generate a plan from an individual's capabilities does not necessarily mean the goal is impossible, since other agents in the society might have complementary capabilities. If an agent is capable of generating new plans at runtime by taking into consideration the capabilities of others, new *multiagent* plans can be used to overcome individual limitations, and added to the plan library for future use. In a simplistic approach, a planning capable agent interpreter that is aware of other agents' capabilities can be used to achieve this effect.

Nevertheless, the ability to overcome problems using cooperation is a key part of a multiagent system, but agent programming languages, designed ostensibly to allow the development of multiagent systems, seldom support high-level cooperation directly. Most programming languages only go as far as including an agent communication language like KQML [Finin et al., 1994] or FIPA ACL [Foundation for Intelligent Physical Agents, 2000], but lack any indication as to how these languages are to be used to achieve any level of agent cooperation. This is clearly a problem, and if agent languages are to be used for agents to cooperate in an interoperable way, these languages need to contain at least a simple cooperation mechanism to facilitate the development of multiagent systems without requiring a designer to create this mechanism from scratch every time. An analogy can be made with object-oriented languages such as Java[1] [Grosso, 2002] or Ruby[2] [Carlson and Richardson, 2006], each of which contains a basic object sharing mechanism on top of some network protocol that allows simple interoperation among distributed instances of the language interpreter. Unlike object-orientation though, cooperation among agents does not involve simply sharing and invoking methods in remote agents. Since the basic building block of an agent description is a plan, agent cooperation must be concerned with the creation of multiagent plans.

Multiagent plans give rise to issues of efficiency and reliability of distributing task achievement. Cooperative action involves communication and coordination, as well as an increased degree of risk for the success of a plan, given that the agents relied upon may break their commitments to achieve their own goals. But from the planner's perspective, the composition of new plans based on preconditions and effects on the environment can be performed independently of these factors. It is important to point out that, though there are a number of issues (*e.g.* communication and coordination among parties) that must be addressed in order to perform planning in distributed systems [desJardins et al., 1999], these can be abstracted away from the planning process and inserted at a later point in time [Ghallab et al., 2004]. In order to address this limitation of agent languages, in this chapter we develop a new technique for agents with plan generation capabilities to cooperate in a multiagent society. Unlike the approach taken in the TÆMS/GPGP platform [Lesser et al., 2004], we do not gear an agent language exclusively for cooperative planning, but adapt an agent language that has been proven for single agent domains and extend it to handle cooperation aspects.

[1]And its Remote Method Invocation (RMI) mechanism.
[2]And its Distributed Ruby (DRb) mechanism.

More specifically, by extracting key information from another agent's plans, particularly in relation to the declarative consequences of local plans, an agent can be informed of the problem-solving capabilities of others, allowing it to delegate the achievement of specific world-states, and using this information in its own planning process. In this chapter, therefore, we extend the planning agent architecture of Chapter 3 to handle multiagent domains, thus addressing the omission of cooperation mechanisms embedded in agent languages. Our contribution in this chapter is twofold: a generic technique to reduce multiagent planning into a traditional planning problem, and the practical integration of such a technique in a BDI-like agent language. In our approach, external plans are encapsulated into patterns of local plans in order to abstract the communication and coordination aspects away from the planner.

To accomplish this, and in keeping with the remainder of this thesis, we choose the AgentSpeak(L) language, so that we can leverage the extensions developed in Chapter 3, and start with an overview of our cooperation mechanism in Section 5.2. Cooperation in agent systems in turn requires a series of fundamental building blocks, and therefore, in Section 5.3 we review our chosen agent communication language, as well as the communication mechanisms and language extensions available for AgentSpeak(L). Having reviewed the technical background of our technique, we develop our cooperation mechanism in Section 5.4, including a partner discovery algorithm for use in our cooperating agent language extension in Section 5.4.1. We further refine the cooperative plans thus created with a failure handling mechanism developed in Section 5.5. Finally, we situate our contribution within the body of related work in Section 5.6, and conclude the chapter with a discussion in Section 5.7.

5.2 Overview of Cooperation in AgentSpeak

As we have seen in Section 5.1, agent languages do not generally include cooperation mechanisms built into them, leaving a significant gap in their potential for the development of complete multiagent systems. Consequently, the problem we need to solve is the inclusion of a cooperation mechanism in an agent language. In particular, the cooperation possibilities among agents that can change their plan libraries at runtime are greater than for agents with pre-designed cooperative behaviour. Since our thesis focuses on *dynamic* plan libraries, the cooperation mechanism we develop in this chapter allows for agents to supplement their plan libraries with new plans based on help from cooperating partners. When an agent has exhausted its individual options to achieve a desire, it may be able to accomplish this desire by relying on cooperation with others. In order to generate new plans that rely on cooperation with others, we define a practical technique for multiagent planning and cooperation that allows an agent to share the knowledge of the consequences of its plans so that others can delegate parts of their high-level plans and achieve new goals. This technique can be divided into three main parts: the discovery of available capabilities, the creation of cooperative plans and the execution of multiagent plans.

When an agent cannot find a plan within its plan library to accomplish its goals, it tries to create a new plan, as we have seen in Chapter 3. However, this process relies only on an agent's own abilities, and in some cases these abilities may not be enough. Since this process does not consider the abilities or plans of other agents, it may still be possible to create appropriate plans that include the plans of other agents, relying on their cooperation. This is the focus of this chapter.

The first task for an agent seeking to do this is to find out if there are plans that other agents are willing to perform for the original agent. We do this by gathering information through a capability discovery mechanism, which provides details of the available plans that other agents may offer to be used in cooperation. Given this information, we can proceed to establish the basis for the creation of cooperative plans. In essence, we want to provide our original agent with extra plans, from the other agents, to integrate into its own plan generation capability, in support of achieving its goals. Now, this is challenging, because we require the *sharer agents* to execute such plans themselves, as part of a broader, higher level plan for the original agent, requiring both communication and coordination. However, we do not want these concerns of communication and coordination to cause difficulties in the plan generation process.

Though many existing approaches to cooperative action handle communication and coordination together with plan composition [Kalofonos and Norman, 2004], we choose to separate these two tasks in order to allow the use of *off-the-shelf* planning algorithms. The rationale behind this choice of approach is as follows:

- as discussed in Chapter 3, there is a wide range of local planning algorithms;
- active research on planning algorithms yields new potentially useful algorithms frequently;
- some planning algorithms are better suited for certain specific domains;
- integrating communication and cooperation in the planning algorithm is not always easy; and
- our approach delegates actions to the agent architecture, allowing new planners to be used seamlessly.

In particular, our technique relies on *plan patterns* that encapsulate communication and coordination in such a way that the planning algorithm can ignore them when chaining operators in a plan.

In our mechanism, the *requester agent* (that is, the agent that needs to use the capabilities of another agent to achieve its goals) needs plans to serve as local placeholders for the invocation of externally executed plans, which we call *proxy plans*. Conversely, the *sharer agent*, that is, the agent willing to execute one of its plans on behalf of another, needs an *external plan* to handle the requests from requester agents and invoke its local plan. As we have seen in Chapter 3, new plans can be generated through classical planning by considering their preconditions and consequences and equating them to STRIPS/PDDL operators, and proxy plans can similarly be integrated into new plans. These newly created *cooperative plans* are now effectively multiagent plans, and can then be integrated into an agent's plan library for future use and efficiency gains.

In order to create these interrelated plans, we use a mechanism inspired by the work of Hübner *et al.* [Hübner et al., 2006] consisting of plan library modification rules (or *plan patterns*) that allow the creation of new plans based on the existing plan library and additional parameters. There are two plan patterns that are needed to generate cooperative plans, without the need for designers themselves to do so.

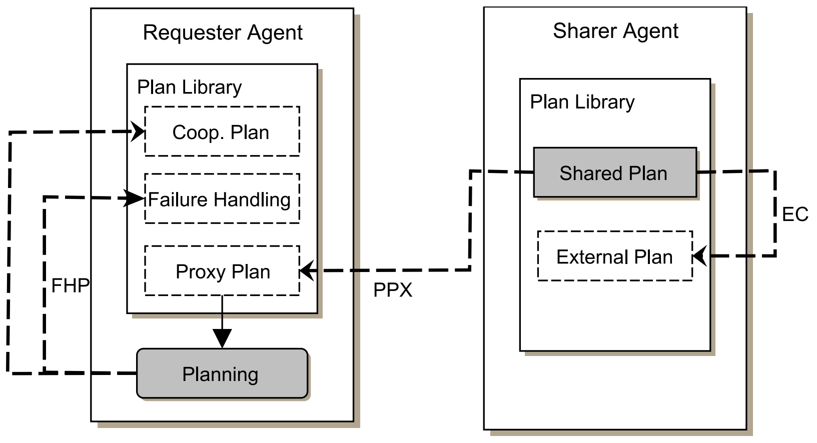


Figure 5.1: Plan patterns involved in the sharing and use of a plan to achieve g.

More specifically, given a *shared plan*, we define an *external plan* (EC) pattern in Section 5.4.4 that includes the steps necessary for another agent to request the execution of the shared plan. On the requester's side, we define a *proxy plan* (PPX) pattern in Section 5.4.3 that encodes the declarative information of the shared plan's preconditions and consequences, and contains the steps necessary for this agent to request the remote execution of the shared plan. Ultimately, proxy plans can be used by the planning process of AgentSpeak(PL) as if they were local plans, and result in new *cooperative plans*.

Now, even if the issues arising from interaction can be ignored by the planner, they must be addressed in order to ensure the long-term effectiveness of the plan library. In particular, a number of issues arise from relying on third parties to accomplish one's goals, such as unreliability and broken commitments. Moreover, it is possible that an agent whose cooperation is necessary for some plan in the plan library may leave the society. This renders such a plan not only useless, but also damaging to an agent's efficiency if the plan is eventually selected to achieve some goal, since this plan will always require the agent to drop it after wasting the effort of starting to execute it. Therefore, given the uncertain nature of agent cooperation, there is also a need to provide a third plan pattern, for *failure handling plans* (FHP) to cope with unreliable partners, which we describe in Section 5.5. All these patterns and their resulting plans are summarised in the diagram of Figure 5.1, where dashed arrows represent the creation of new plans through a plan pattern.

5.3 Technical Requirements for Cooperation

We have seen in Section 2.5.1 that agent communication is generally underpinned by speech act theory, but the original AgentSpeak(L) does not include communication constructs for agent communication of any sort. In order to address this shortcoming, we leverage the work of Moreira *et al.* [Moreira et al., 2003], who introduce a notation for speech act-based communication into AgentSpeak(L). This work was later extended by Vieira *et al.* [Vieira et al., 2007] formalising the semantics of the main KQML speech acts within the AgentSpeak(L) reasoning cycle. In this section, we review the relevant aspects of their work to provide the theoretical basis for communication in our cooperation technique. Agent communication as defined by Vieira *et al.* [Vieira et al., 2007] is underpinned by two notational extensions to AgentSpeak(L), *annotations* and *internal actions*, which we review in Sections 5.3.2 and 5.3.1 respectively. Finally, we review the semantics of the key speech act performatives required for our technique in Section 5.3.3.

5.3.1 Internal actions

The common understanding of agent actions is that they are environment transformation operators, so that when an agent invokes an action, some consequence in the environment is expected. However, when some custom computation needs to take place within a single reasoning cycle, Bordini *et al.* use the concept of an *internal action* in AgentSpeak(XL) [Bordini et al., 2002]. This allows an agent to access extensible libraries of custom procedures that can be executed instantaneously by an agent.

Unlike traditional actions, internal actions do not cause changes in the environment, and since they are executed instantaneously, they can be included in either the body or the context of a plan, allowing an agent to use them to refine the process of selecting applicable plans. Syntactically, internal actions are denoted in the language by a preceding dot (.) character, so if a designer wants to define a *check* internal action with two parameters, its invocation is represented as $.check(a, b)$.

From a communication point of view, an internal action *.send* is invoked by an agent whenever it needs to communicate through some speech act performative to another agent.

We further explore the effects of this action depending on the specific performative in Section 5.3.3

5.3.2 Annotations

As part of the formalisation of communication within AgentSpeak(L), Vieira *et al.* [Vieira et al., 2007] introduce a modification to the language to allow *annotations* to be associated with predicates within AgentSpeak(L). These annotations were initially proposed by Bordini *et al.* [Bordini et al., 2002] and are aimed at extending the representation of beliefs from simple predicates to predicates with an indication of their source. Syntactically, annotations are represented as a list of additional predicates appended to the annotated predicate. For example, while in traditional AgentSpeak(L), we represent a belief in some fact `p(X)` as a predicate `p(a)`, their extension allows us to represent that an agent believes in `p(a)` because it has perceived it through its sensors and because some other agent `ag1` has also informed our agent that `p(a)` is true, as `p(a)[source(percept), source(ag1)]`.

Along with the syntactic modification, Vieira *et al.* [Vieira et al., 2007] also extend the meaning of logical consequence and unification between annotated predicates, stating that an annotated atomic formula $p_1[s_{11}, \ldots, s_{1n}]$ is a logical consequence of a set of ground atomic formulae bs, if and only if there exists $p2[s_{21}, \ldots, s_{2m}]$ in bs such that p_1 and p_2 unify and $[s_{11}, \ldots, s_{1n}]$ is a subset of $[s_{21}, \ldots, s_{2m}]$.

This annotation mechanism allows triggering events, which we have seen in Section 3.3.1.2 are formed by predicates referring to beliefs or goals, to encode information regarding their source, thus enabling plans to handle events resulting from communication in a richer way. Intuitively, from the notion of consequence explained above, one annotated predicate is only supported by another predicate that unifies with it and that has at least the same set of corroborating sources as the original predicate. For example, a certain predicate `p(a)[source(ag1)]` is supported by predicate `p(X)[source(ag1),source(ag2)]`, but not by predicate `p(X)[source(ag2)]`.

In our work in this chapter, we use annotations in the messages exchanged between cooperating agents to indicate the sender of each message. This information is used by agents sharing plans in their decision over whether to agree to cooperate with the requesting agents.

5.3.3 Speech act-based communication

Effective cooperation between autonomous agents in a society requires some form of communication among them, and research into agent communication has generated a number of agent communication languages, such as FIPA ACL and KQML [Singh, 1998]. As we have seen at the beginning of this section, Vieira *et al.* [Vieira et al., 2007] introduced an operational semantics of speech act-based communication for AgentSpeak(L), defining several plan rules for handling several of the performatives defined by Searle [Searle, 1969]. The plan rules thus defined are given from both a sender and a receiver point of view, allowing them to be implemented in practical AgentSpeak(L) interpreters.

From an operational perspective, we consider an implementation of agent message passing using the concept of *internal actions* described in Section 5.3.1, because messages between agents are not expected otherwise to cause the environment to change. Messages are therefore sent using the *.send* internal action, which takes three parameters: the identification of the receiver, the performative carried by the message, and the message itself. In order to operationalise the effect of sending and receiving messages to and from other agents, Vieira *et al.* [Vieira et al., 2007] extend the configuration of an AgentSpeak agent with two new data structures: a set of messages that an agent has received but has not processed (effectively an *inbox*), and a set of messages that need to be sent to other agents (effectively an *outbox*). When the send internal action is invoked, messages are stored in the outbox until the interpreter moves them to the receiver's inbox. On the receiver's side, the inbox effectively provides another set of events, just like traditional events from the environment, to which an agent responds by adopting plans. Thus, from a plan execution perspective, the receipt of new messages is handled in the exact same way as the receipt of new events from the environment.

We now proceed to describing the effects of sending and receiving messages within the AgentSpeak implementation of KQML. In particular, in this chapter we are concerned with the three performatives needed for our cooperation technique:

- *ask*, used by an agent to request information from others;
- *tell*, used by an agent to supply information to others; and
- *achieve*, used by an agent to request another agent to achieve a procedural goal.

According to Vieira *et al.* [Vieira et al., 2007], there are two possible effects on the sending agent of executing such an internal action, one for the *ask* speech act, and another for all other speech acts. When an agent sends a message with the *ask* performative, it is trying to check whether or not another agent has a certain belief within its belief base; in effect, an agent is trying to execute a query in the target agent. Therefore the intention containing this send action must be interrupted until the query has been performed, or the receiving agent has rejected the message. The other possible effect of executing a send action (*i.e.* for tell and receive), is that a message is sent to the target agent and the intention resumes executing. We proceed to examining the effects of these messages in the receiving agent.

For the target agent, messages are ultimately received as events annotated with the name of the sending agent, so the query associated with the *ask* performative being sent from an agent *ag* results in an event +?query[source(ag)] being posted for the receiving agent.

For example, suppose agent *randal* wants to know the time of the hockey game, stored in the belief base of agent *dante* as the belief **time**(hockey,T). To discover this information, it executes an internal action .send(dante,ask,time(hockey,T)), which causes an event +?time (hockey,T)[source(randal)] to be posted to *dante*. If *dante* accepts the message, and has **time**(hockey,1020) in its belief base, the .send action in *randal* will be executed successfully, resulting in T being unified with 1020. Notice here that since the effect of this send is an event in the receiving agent, it might be handled by a plan with a triggering event matching the query being made to it, rather than a direct query to its belief base.

It is important to note that annotations also include reserved words indicating that the source of certain predicates are an agent's sensors, denoted as source(percept), as well as indicating that certain events originate in the agent itself, denoted as source(self). This enables an agent to maintain its autonomy, allowing plans to be created to respond to specific agents, or not to respond to anyone at all. For example, if a designer wishes to create an agent that never responds to queries from other agents, a plan +?Q[source(S)] **:** **not** (S=self) <− false. can be included in the plan library.

When an agent sends a message with a *tell* performative, it wants to make another agent aware of some belief expression. Thus, when an agent receives a message with a tell performative from an agent *ag* regarding a belief *b*, the target agent receives an event +b[source(ag)]. For example, now suppose *dante* wants to make *randal* aware that the hockey game is at 1020 by executing the action .send(randal,tell,time (hockey,1020)). If *randal* accepts this message, it causes the event +time(hockey,1020) [source(dante)] to be posted to *randal*.

Finally, when an agent wants another agent to adopt a particular achievement goal, it sends a message with an *achieve* performative. Similarly to the previous types of performatives, the effect of an agent *ag* sending a message with the achieve performative regarding a goal *g* is that an event +!g[source(ag)] is posted to the receiving agent. So, following our previous example, if *dante* wants *randal* to come to the hockey game now, and it knows that *randal* has a plan to come to the game associated with the triggering event +!comeToHockey, it executes .send(randal,achieve,comeToHockey). Again, if *randal* accepts this message, it causes the aforementioned event to be posted to *randal*, and the plan to be executed.

5.4 A multiagent planning mechanism

In this section, we explain our multiagent planning mechanism. We start by providing a basic capability discovery mechanism in Section 5.4.1, which gathers information about willing cooperation partners and the ability they are trying to share. Next, in Section 5.4.2 we explain how we use the plan patterns mechanism to implement our cooperation mechanism. We use this mechanism to build key parts of the cooperation mechanism, including proxy plans (in Section 5.4.3), and external plans (in Section 5.4.4). Finally, in Section 5.4.5 we show how multiagent plans created locally are executed and how potential failures of these plans can be handled.

5.4.1 Discovering Capabilities

In order for an agent to be able to use the capabilities of others, it must be aware of what capabilities are *shared* by whom. Here, a shared capability is a plan that an agent is willing to execute on behalf of another agent whose goals cannot be accomplished by its individual abilities. In turn, for an agent to be able to use the plans of others rather than just its own, it must seek partners willing to carry out some of their plans on behalf of the requesting agent. These willing partners then send declarative information about their plans. Since we are not concerned in this thesis with issues of discovery per se, we avoid developing an elaborate mechanism, and use a dummy plan (or a plan *stub*) indeed as an abstraction of any of a number of existing partner selection mechanisms, such as described in [Munroe et al., 2004].

Nevertheless, we provide a simple capability discovering mechanism in which we assume that cooperation partners have already been selected somehow. Our capability discovery method consists of broadcasting a request for external plans, which is answered by all available agents in the society. However, if a partner selection mechanism is in place, the requests for external plans will only be sent to selected cooperation partners. During the process of gathering information regarding shared capabilities, plans comprising the abstraction layer for the planner are created in both the agent that will eventually use the shared capability in its planning process and the agent providing the capability. This mechanism is shown in Algorithm 4

Algorithm 4 Simple capability discovery mechanism.

```
Require: Set $P = \{ag_1, \ldots, ag_n\}$ of selected cooperation partners
  Broadcast to $P$ a request for capabilities
  Receive a set $C_P = \{c_{1,ag_1}, \ldots, c_{m,ag_1}, \ldots c_{o,ag_n}\}$ of capabilities
  for all $c_{i,ag_j} \in C_P$ do
    Create local plans to use $c_{i,ag_j}$
    Create plans in $ag_j$ to share $c_{i,ag_j}$
  end for
```

Elements in the set C_P must contain all the information necessary for the requesting agent to perform planning as if the capability being shared was local. Thus, similarly to the way in which information about preconditions and effects is collected for planning in AgentSpeak(PL), partners wishing to inform others of their external plans need to gather the plan invocation parameters, preconditions and declarative effects and send this information to their peers. This information can be retrieved using the same process as in AgentSpeak(PL), but instead of using this information to generate a STRIPS-like operator description, an agent sends this as a reply to another agent requesting external plans, along with the identification of the agent supplying the external plan. Therefore, elements c_{i,ag_j} contain the information represented in the tuple, $\langle g, ag, P, E \rangle$, where:

- g is the achievement goal (including its parameters) in the sharing agent's plan library that will be executed on behalf of the requesting agent;
- ag is the identifier of the agent that contains the external plan;
- P is a set $\{p_0, \ldots, p_n\}$ of preconditions g; and
- E is a set $\{r_0, \ldots, r_m\}$ of declarative effects expected to hold after the external plan is executed.

This information is used in the creation of a series of plans constituting the abstraction layer. This information and the plans resulting from it, in turn, enable an agent to generate new high-level cooperative plans, which we detail in Section 5.4.5.

5.4.2 Plan patterns

While many researchers have chosen to create new languages to add notions such as declarative goals [Sardiña et al., 2006; Winikoff et al., 2002] and failure handling mechanisms [Sardiña and Padgham, 2007; Thangarajah et al., 2007], it is possible to represent these, and many other notions using simpler, existing agent languages. In traditional AgentSpeak(L), although there is no explicit connection between the adoption of plans and the goals they achieve, we have made the connection explicit through the planning action of AgentSpeak(PL) in Chapter 3. Another alternative, in AgentSpeak(L), is to represent these notions by multiple related plans, as shown by Hübner *et al.* [Hübner et al., 2006], who introduce the notion of *plan patterns* to facilitate the designer's task of creating multiple, related, plans that serve a particular purpose.

The idea here is that many related plans to accomplish a particular task will have reoccurring elements, varying just in small details that can be parameterised. Such parameters are more complex than what could be achieved through the unification mechanism present in an AgentSpeak interpreter like Jason. For example, a high-level plan to try multiple different plans in sequence to achieve a world-state, checking after plan execution if the world-state has been achieved can be seen as a plan pattern where the preconditions and the test at the end of each plan are appended to each of the plan in the series [Hübner et al., 2006].

In more detail, a plan pattern is an agent program rewriting rule with a numerator describing the original plan (or plans) description, and a denominator describing the resulting agent program. For example, if we wish to define a plan pattern that adds a printed message before and after a certain plan body b is executed, called **PDB** (Plan Debug), the rule is defined as:

$$\frac{+e : c \leftarrow b.}{\begin{array}{rl} +e: & c \\ \leftarrow & .print(\text{“}Start\text{”}); \\ & b\,; \\ & .print(\text{“}End\text{”}). \end{array}} \textbf{PDB}$$

Pattern **PDB** takes a plan with a certain plan with a triggering event e, a context condition c and a body b, and replaces this plan in the plan library by another one with the exact same header (*i.e.* with e and c), but with a body containing a step to print the word “Start” before executing the original plan body b, and prints the word “End” after executing the original plan body.

5.4.3 Creating proxy plans

Once an agent is aware of the external plans of others in the same environment, it can try to use these capabilities in its own problem-solving strategies. In this approach, we make the *external* aspect of *shared plans* transparent to an agent's local planner through *proxy plans*. These proxy plans describe the expected outcome of a successful invocation of a third party capability and encapsulate the communication and coordination necessary for effective cooperation. A proxy plan pattern **PPX** for an external plan $\langle g, a, P, E\rangle$, where g is the triggering event of the shared plan in the sharing agent a, $P = \{p_0, \ldots, p_n\}$ is the set of preconditions (*i.e.* the context condition) of the shared plan, and $E = \{r_0, \ldots, r_m\}$ is the set of declarative effects of the shared plan, is as follows:

$$\frac{+!g : p_0\& \ldots \& p_n \leftarrow r_0; \ldots ; r_m}{\begin{array}{rl} +!remoteG: & p_0\& \ldots \\ & \& p_n \& ready(a, g) \\ \leftarrow & .send(a, achieve, \\ & \quad requestG); \\ & .wait(done(g)); \\ & +r_0; \ldots ; +r_m. \\ +!check(a, g): & true \\ \leftarrow & +ready(a, g). \end{array}} \mathbf{PPX}_{g,a,P,E}$$

This plan pattern creates two plans, one of which replicates all the logical constraints required for a to be successful in executing this plan locally. The plan body includes a communication action (*.send*) that uses the *achieve* performative to request the sharing agent to carry out the specified plan, followed by an action to wait for confirmation that the plan was executed. Finally, the plan pattern replicates the belief additions expressed in the sharing agent's external plan, so that the planning process of AgentSpeak(PL) [Meneguzzi and Luck, 2007a] can process this plan in the same way as it would process local plans.

In addition to the action-related part of the proxy plan to invoke the external plan, we may also want to check that the owner of the external plan is ready and willing to adopt it. This is represented in the **PPX** plan pattern by the precondition $ready(a, g)$, which is added to those preconditions already present in the original external plan. This literal, in turn, is the result of an additional plan to ensure that the sharing agent will actually carry out that action when the requesting agent needs it to do so. In the **PPX** plan pattern, this plan is simply a placeholder for any mechanism used to ascertain the reliability or readiness of a cooperation partner, which can be replaced by any mechanism preferred by the designer. Such a mechanism can be introduced using the **CA** (check agent) plan pattern, which rewrites the *check* plan so that it calls a plan in the plan library associated with such mechanism. For example, if there is a trust verification mechanism associated with a *verifyTrust* achievement goal (which we will not specify, but assume to be specified by the designer), a plan pattern **CA** for the readiness of an agent to execute external plan $\langle g, a, P, E\rangle$ through an achievement goal *verifyTrust* is as follows:

$$\frac{+!check(a,g) : true \leftarrow +ready(a,g).}{\begin{array}{rl} +!check(a,g): & true \\ \leftarrow & !verifyTrust(a,g); \\ & +ready(a,g). \end{array}}\,\mathbf{CA}_{g,a}$$

Now, let us consider an example where we have two agents, *randal* and *dante*, both of which work in a convenience store, and can close it at the end of the day [Smith, 1995]. Both their plan libraries should include a plan that accomplishes this, and the plan illustrated in Table 5.1 shows Randal's plan to close the store.

```
+!close(store) : at(randal, store)
   <- .switchOff(lights,store);
      .close(door,store);
      +closed(store).
```

Table 5.1: Randal's plan to close the store.

Normally, Dante is assigned the last shift and closes the store, but on a certain day, Dante needs to leave earlier to go to an appointment, so he needs somebody else to close the store on his behalf. In order for Dante to be able to incorporate this ability in his plans for the day, Dante needs to know the information from the shared plan of Table 5.1 in order to use the **CA** plan pattern, including the preconditions and effects. Considering the rules for extracting declarative information from Chapter 3, we have that the precondition of this plan is at(randal,store) and the effect is closed(store). By applying the plan pattern **CA**, we have the resulting proxy plan of Table 5.2.

```
+!remoteClose(store) : at(randal, store) & ready(randal,close(
    store))
  <- .send(randal, achieve, requestClose(store));
     .wait(done(close(store)));
     +closed(store).
```

TABLE 5.2: Dante's proxy plan to close the store through Randal.

With this plan in his plan library, Dante can now plan for his early departure from the store, but we also need Randal to be aware of how to respond to Dante's request. For this we also need to change Randal's plan library, creating plans to communicate with Randal.

5.4.4 Creating external plans

An important property of our proxy plans is that they succeed when the sharer agent succeeds, and fail if either the sharer agent fails in its execution or it refuses to carry out its commitment. Hence, from the requester agent interpreter's point of view, the execution of a local plan and an external plan is the same.

Naturally, an agent sharing an external plan needs to have in its plan library the achievement goal that corresponds to the *achieve* performative sent by the requesting agent. We refer to this achievement goal as a *plan endpoint* to the **PPX** plan pattern, which is associated with an actual plan in the sharing agent's plan library. The external plan, is therefore generated from a local plan in the sharer's plan library using the **EP** (external plan) pattern, which is as follows:

$$\frac{+!g : e \leftarrow b.}{\begin{array}{rl} +!g : e & \leftarrow b. \\ +!requestG & [source(S)] \\ : & true \\ \leftarrow & !g; \\ & .send(S, tell, done(g)). \end{array}}\,\mathbf{EP}_g$$

Following the example developed in Section 5.4.3, we need to modify Randal's plan library in order to allow his plan to close the store to be accessed by Dante. As we have seen, Dante's proxy plan sends a request for an externally invocable plan to be executed by Randal. This plan can be created through the application of the **EP** plan pattern to the plan of Table 5.1, which keeps the original plan in the plan library and adds a plan that receives the request from Dante and executes the shared plan, as illustrated in Table 5.3.

```
+!requestClose(store) [source(S)] : true
   <- !close(store);
      .send(S,tell,done(close(store))).
```

Table 5.3: External plan for Randal to close the store.

5.4.5 Creating cooperative plans

Given the properties of the proxy plans described in Section 5.4.3, it is easy to use the planning approach of AgentSpeak(PL) to generate new multiagent plans, since the AgentSpeak(PL) planning module is insulated from the communication and cooperation aspects of planning. However, although the generation of a sequence of actions (from a cause and effect perspective) does not depend directly on whether it includes external and internal capabilities, high-level plans that depend on the compliance of third parties must contain guards to prevent initiating the plan when it has become infeasible. These guards are derived by propagating the preconditions of external proxy plans to the precondition of the high-level plan generated by the planning module. Propagating these preconditions ensures that a plan will not be initiated until all parties are ready to comply with requests for cooperation, while making sure that the cooperating agent is queried for availability immediately before its cooperation is needed. This can be accomplished using the algorithm for precondition generation developed for AgentSpeak(PL) in Section 3.6.3.

Now, to finish our example, suppose that Dante is aware that Randal can achieve a goal to close the store on his behalf. Moreover, if Dante needs Randal to close the store on his behalf, the plan to request Randal to be at the store requires him to be at the store, and results in the store being closed; a cooperative plan to achieve these goals generated in our system is shown in Table 5.4.

```
+!goal_conj([closed(store)]) : at(randal, store) & ready(randal)
   <- !remoteClose(store).
```

Table 5.4: A cooperative plan.

When a cooperative plan is adopted by an agent, it eventually reaches the step corresponding to the adoption of the proxy plan (*remoteG*). The proxy plan causes this agent to send a message to the sharer agent requesting it to execute its external plan (*requestG*), which corresponds to delegating the adoption of a plan to achieve a certain goal g in the sharer agent's plan library. If the plan to achieve g is executed successfully, the sharer agent sends the confirmation of having achieved g. This sequence of events is illustrated in Figure 5.2.

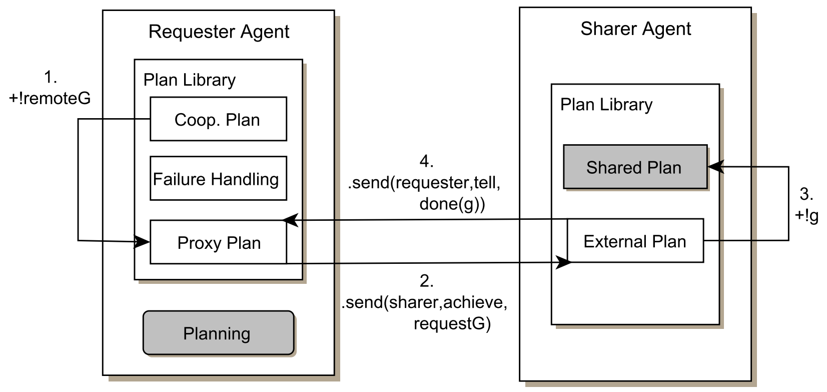


FIGURE 5.2: Proxy plan communication.

5.5 Failure handling for new plans

As we have seen, it is often pointed out that traditional agent architectures equate goal achievement with the successful execution of a procedural plan selected to achieve such a goal [Hübner et al., 2006; Winikoff et al., 2002]. Here, the assumption is that the invocation condition of a procedural plan should be enough to guarantee that the plan executes successfully in case the goal is possible, or proves the goal impossible otherwise. However, in highly dynamic environments, in particular those populated by self-interested agents, the conditions that enabled a plan to be executed may change after an agent commits to a plan. Our technique for multiagent cooperation must take into consideration the unreliability of cooperation in the context of self-interested and unreliable agents by adding failure handling plans to manage multiagent plans, and eventually to remove plans including such unreliable partners. Therefore, in order for agents to be able to cope with the possibility of plan failure, they must be able to handle failure explicitly. Recent work in agent languages has explored the semantics of plan failure, extending traditional languages with plan failure constructs [Bordini et al., 2005b] as well as procedures for handling plan abortion [Thangarajah et al., 2007].

Although the ability to create new plans taking advantage of the external plans of other agents allows the creation of plans that achieve goals otherwise impossible for an agent, dependence on other self-interested agents poses another challenge, coping with possibly unreliable partners. Plans created at design time tend to be very efficient by making assumptions about aspects of the environment that do not change at runtime, whereas the generation of plans at runtime involves a great deal of computational effort. We have seen in Chapter 3 that the computational effort of planning at runtime can be amortised by the careful generation of contextual information and storage of newly created plans in the plan library. However, plans created in a dynamic society in which autonomous agents may join and leave at any point in time cannot make many assumptions regarding the availability of capabilities shared by third parties. The likelihood of failure for plans that depend on others can thus be considered greater than for plans that rely on an individual's own capabilities. Therefore, it is necessary for dynamically generated plans, especially those that depend on unreliable capabilities, to have associated failure handling plans. This mechanism of failure handling relies on the notion of a plan-dropping event described in Section 3.4.6. For example, if an agent creates a plan that involves cooperation with an agent a, we introduce a *failure handling plan* **FHP** pattern that removes the failed plan when this agent fails to cooperate for some reason, as follows:

$$\frac{+!goal\_conj([g_1, \ldots, g_n]) : e \leftarrow b.}{\begin{array}{rl} +!goal\_conj([g_1, \ldots, g_n]) & \\ & : e \leftarrow b. \\ -!goal\_conj([g_1, \ldots, g_n]) & \\ & : notready(a) \\ \leftarrow & .remove\_plan(\\ & \quad goal\_conj([g_1, \ldots, g_n])). \end{array}} \, \mathbf{FHP}_a$$

Here, handling plan failures is important to ensure that the agent can cope with individual faults due to failed cooperation. It also allows the agent to manage its plan library in the long term, removing plans which are no longer relevant due to the absence, or consistent lack of reliability, of necessary parties.

5.6 Related Work

Previous work by Ancona *et al.* [Ancona et al., 2004] describes a cooperation technique that allows agents to expand their problem solving capabilities at runtime by exchanging plans at runtime. Although this technique relies on a very similar basic agent framework to our own (aside from the planning component), it consists of a distinct approach to addressing the shortcomings of an agent, as it relies on an agent receiving entire plans from others. This approach, therefore, assumes that all agents in an environment are able to execute the same set of basic actions, which may not be the case in many real world scenarios. For example, agents might require different levels of authorisation to perform specific actions in the environment: an agent running in a user-level account, performing maintenance in a Unix filesystem may need to change a file that is owned by the root user, and clearly the plans that the root can execute cannot simply be sent to this agent. Ancona's approach is complementary to ours in the sense that it can, for example, replace the planning module we use to generate new plans from scratch and allow an agent to get new plans from others.

Though planning has always been advocated as a fundamental part of an agent system, it is only recently that consistent efforts have been made to integrate AI planning into agent architectures as a result of significant improvements in the runtime efficiency of modern planning algorithms. These efforts have largely focused on integrating planning into individual agents by relying on a centralised planning algorithm to solve specific problems using a state-space planner [Despouys and Ingrand, 2000], or to check the viability of existing partial plans by verifying them before adopting these plans [Sardiña et al., 2006]. Meanwhile, distributed planning has evolved more or less in parallel with multiagent technology, with very few intersecting efforts [desJardins et al., 1999]. While research on distributed planning focuses mostly on distributing the burden of planning algorithms among a number of contributing nodes, multiagent cooperation is mainly concerned with coordinating pre-defined plans among participants whose existence and potential are known before the system is deployed. This parallel progression of planning versus agents is summarised in Figure 5.3.

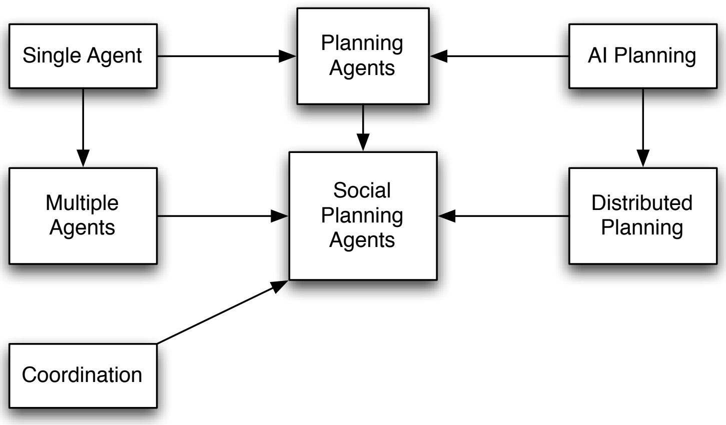


FIGURE 5.3: Evolution of planning and agent technology.

In a dynamic environment with multiple autonomous agents, it is important not only to coordinate predefined plans, but also to create and coordinate new strategies. Multiple agents need to detect and discover how to exploit unforeseen combinations of partners and capabilities while at the same time coordinating their efforts to minimise conflicts (which may arise due to agents having diverging individual goals) and maximise utility. Reasoning about cooperation in a dynamic world presents a number of challenges that have been addressed separately in previous efforts, namely:

- deciding when to cooperate and with whom;
- discovering and exploiting capabilities among participants;
- coordinating efforts that require synchronisation;
- reaching a consensus among agents with distinct goals;
- determining the ideal amount of information sharing; and
- coping with a changing world during plan formation.

We further explore recent efforts in distributed planning and its integration with agent systems, discussing their advantages and limitations at the end of this section. These efforts illustrate the possibility of many interesting extensions to our technique.

5.6.1 MARTHA Project

The MARTHA project, (Multiple Autonomous Robots for Transport and Handling Applications) [Alami et al., 1998], created a PRS-based multiagent system aimed at controlling a fleet of mobile robots in charge of running trans-shipment operations (*i.e.* loading and unloading containers) in harbours, airports and marshalling yards. The system is composed of a central station (CS) that plans trans-shipment operations and assigns missions to agents running in the mobile robots.

Communication between the CS and the robots is minimal, and the CS simply tells robots their missions and preferential routes, while they must coordinate individually with other agents. Coordination in MARTHA is exclusively concerned with the avoidance of resource conflicts. Here, resources are sections of the routes over which robots navigate. Individual robots generate plans to accomplish their assigned missions, but before committing to the execution of these plans, they attempt to *merge* them into a global plan. In order to merge its plan, a robot needs permission from the other agents, in the form of a *token* passed around agents, as well as information on the usage of shared resources. If an agent manages to successfully merge its plan into the global plan, it then requests other agents for synchronisation notifications to solve any potential conflict during execution. For instance, if two agents need to pass through the same intersection, the agent allowed to pass first would have to send a message to the second one notifying it of its turn. Planning in this system consists mainly of plotting trajectories within a set of established routes, which are divided into cells representing the building blocks for the movement plans.

Although the approach taken in MARTHA for support to cooperation in a multiagent system is interesting in that it uses planning to create new strategies at runtime, this work has a very narrow scope, and thus lacks the generality afforded by the technique developed in this chapter.

5.6.2 Plan selection through trust

The control cycle of traditional BDI architectures seldom, if ever, address the issues of how to cooperate and with whom. Therefore, Griffiths and Luck [Griffiths and Luck, 1999] extend a BDI-like architecture with higher-level control mechanisms aimed at improving plan selection in a multiagent environment, and consider how cooperative plans are structured by extending the elements of BDI plans. These plans include individual actions, which are the traditional components of BDI plans, as well as *joint actions* and *concurrent actions.*

Joint actions are composite actions that require a group of agents to act simultaneously, or with a precise level of synchronism, whereas parallel actions are cooperative in the sense that a group of agents must contribute to it, but without any particular synchronism.

BDI plans use partial plans that include subgoals instantiated at runtime, and plan selection involves completing these subgoals with a choice of plans. Since plans may or may not contain cooperative actions, the decision to cooperate would normally be implied by the choice of a plan containing such actions.

Previous work on cooperative problem solving does not properly address why agents decide to seek assistance beyond a high level statement that an agent detects the need to cooperate. Griffiths and Luck introduce a series of plan evaluation criteria to help improve plan selection. These criteria involve evaluating not only the quality of a plan in terms of its cost and agent's preference, but also the risk involved in depending on external help. In assessing a plan's risk, an agent builds a model of each partner which include their capabilities and a level of trust in them. All these metrics are brought together in the form of a plan rating, which is calculated in a pre-execution step that simulates how a plan would work when executed taking into account an agent's own model, as well as the inspected models of all other agents involved. This simulation is performed offline, so if new agents and capabilities can be introduced into the world at runtime, this approach would have to be revised.

The work of Griffiths and Luck uses the existing BDI-based plan execution capabilities to reason about cooperative plans, but it assumes cooperative plans are already present in a plan library. Alternatively, our work allows such multiagent plans to be generated from an existing single agent library, but we do not go into much detail regarding the decision to use a multiagent plan, and assume that an agent uses such plans when it has no other alternative. Thus, Griffiths and Luck's work can be seen as complementary to ours in providing a more elaborate process to decide on when to use cooperative plans.

5.6.3 Multiagent Graphplan

Kalofonos and Norman [Kalofonos and Norman, 2004] propose a multiagent planning algorithm based on a version of Graphplan that interleaves planning with communication and allows the negotiation of team goals during Graphplan's solution extraction phase. This planning strategy is extended to allow agents to interleave planning graph expansion with information exchange, as well as interleaving solution extraction with conflict resolution. Information sharing in the graph construction phase is not necessarily complete, and can be limited by a participant's willingness to contribute certain actions to a team plan, hence agents can withhold their unrelated activities from others in this particular interaction.

Some restrictive assumptions are made about the agent's knowledge about the world and the effects of actions, namely that agents agree on the relevant aspects of the initial state, and about the preconditions and effects of actions. Agents also agree on the team goals and are assumed to be cooperative.

During planning, each agent keeps a copy of the planning graph and shares updates with other agents. When each action level is built, agents reveal the actions they are willing to contribute at that point in time. In order to preserve Graphplan's guarantee of termination, once an action is committed to a plan, it cannot be withdrawn in further action levels. Finally, during solution extraction, agents collaborate to search for a plan that represents a compromise to all participants.

Since the work presented in this chapter relies on centralised planning for multiagent plans, we do not take advantage of the potential distribution of planning effort. In this respect, the work of Kalofonos and Norman represents a potential extension to our cooperative planning framework if distributed planning is desirable.

5.7 Conclusion and Discussion

By taking advantage of recent developments in practical agent languages, we have described a practical, yet flexible, technique for multiagent planning. This technique extends the work in Chapter 3 to take advantage of the availability of cooperating agents in a society, allowing agents to overcome individual limitations by delegating *parts* of locally generated plans for execution by others. Furthermore, we have shown how this technique can be implemented using recent extensions to the AgentSpeak(L) language, without affecting the generality of our approach, since any other BDI-like language with declarative goals and communication capabilities can be extended with the planning we propose.

Examples of agent languages suitable for implementing this strategy are CANPLAN2 [Sardiña and Padgham, 2007] and Jason [Bordini et al., 2005a]. Descriptions of agent plans throughout this section use Jason to facilitate readability and to remain consistent with the remainder of this thesis, but these plan definitions can be easily converted to any BDI-like agent language. Jason is an AgentSpeak(L) interpreter with a number of extensions necessary for our technique to function in practice.

We have implemented this cooperation technique using the Jason interpreter extended with the AgentSpeak(PL) functionality of Chapter 3. The resulting system was then used in implementing the example of the convenience store clerks developed throughout this chapter[3]. In practice, a designer using our prototype needs to mark all *shared plans* in the plan library definition as such, while the plan pattern mechanism present in Jason expands these plans at runtime into the corresponding *external plans*. Similarly, during the execution of the capability discovery mechanism of Algorithm 4, the information regarding external plans

[3]This implementation is available for download at www.meneguzzi.eu/felipe/software.html.

from other agents is injected into the Jason interpreter as plan patterns that are expanded into the *proxy plans* used by the planner.

Unlike the previous contributions, the introduction in this chapter of a cooperation mechanism in AgentSpeak(L) is diffcult to evaluate quantitatively, or to compare quantitatively with the original architecture. In consequence, our validation effort consists of example applications of the techniques developed, demonstrating what can be achieved using them. Further evaluation of the contribution in this chapter can and should be undertaken through extensive deployment and use, with experience feeding into evaluation and subsequent refinement. While this is beyond the scope of what is possible in this thesis, it offers opportunities for further research.

Our focus here is on the structural and functional aspects of the plan library, and as a consequence we have sidestepped any detailed account of how to address two major issues with cooperation in agents, namely the distribution of the planning effort, and the evaluation of reliability of cooperation partners. We chose, however, to modularise our technique so that a designer can choose from the existing body of work in both these areas. Regarding the issue of distribution, although the classical planning module leveraged from AgentSpeak(PL) [Meneguzzi and Luck, 2007a] is a simple and centralised one, we see no hurdles in using our technique with distributed plan formation algorithms, such as that proposed by Zhang *et al.* [Zhang et al., 2007]. In this respect, our method is flexible in that it allows any planning algorithm with a PDDL [Fox and Long, 2003] compatible planner to be used in the planning module. Moreover, we acknowledge that issues of trust and reliability of cooperation partners are of paramount importance in any deployment of a system composed of agents that use our technique, but this is not the focus of our thesis, and can be isolated from the rest of our planning process.

Chapter 6

Normative processing

6.1 Introduction

In Chapter 5 we developed a mechanism to extend AgentSpeak(L) beyond only single agents to provide mechanisms to handle societal aspects of multiagent systems. Yet in a multiagent society containing autonomous self-interested agents we need to ensure that agents operate effectively and find ways to work together. In many cases this is achieved through imposing constraints on agents, aimed at maintaining stability in the system. In this context, there have been several efforts to develop computational systems of societal norms [Lopez y Lopez et al., 2005; Vázquez-Salceda et al., 2005], their processing and monitoring for compliance. However, while there has been much work on formalising such systems, very little has been proposed in terms of practical, agent language-level mechanisms to modify an agent's behaviour at runtime (rather than by design) so as to comply with these norms.

In particular, systems composed of heterogeneous autonomous agents require some form of societal control to ensure a desirable social order in which agents work together effectively. For many, *norms* are the mechanism of choice to address this concern in multiagent societies and ensure order and predictability [Aldewereld et al., 2006]. Such norms define standards of behaviour that are acceptable in a society, indicating desirable behaviours that should be carried out, as well as undesirable behaviours that should be avoided. Normative systems thus rely on a representation of obligations, prohibitions and permissions that ensures that complying agents act within some predefined bounds. Although deontic concepts have received much attention from philosophy [Jones and Pörn, 1986; Pörn, 1970], and more recently computer science [Lomuscio and Sergot, 2003; Lopez y Lopez et al., 2005], we must provide a simple definition for the concepts we use in this thesis. Therefore, we take an obligation to be a *positive* constraint on an agent, indicating that it *must act* to accomplish something in the world [Oren et al., 2008a]; permissions to be overriders of obligations, undercutting them and freeing an agent from being bound to a particular constraint; and prohibitions to be negative constraints on an agent, indicating that it must *refrain* from acting in a particular way [Oren et al., 2008a]. The specification and maintenance of normative systems has been the focus of recent research in agents, for example in the context of electronic institutions [Garcia-Camino et al., 2005]. However, these efforts have been largely at the macro level, such as managing global sets of norms in an environment [Vasconcelos et al., 2007] and monitoring an agent's actions within a society for compliance [Faci et al., 2008], among others. By contrast, little work has been done on dealing with the desired *effects* of norms at the level of an individual agent, and traditional agent architectures are generally lacking in mechanisms to adapt agents to comply with newly perceived norms at runtime. In practice, agents operating under normative restrictions are *designed* so that they do not violate these restrictions, and if they do, this must be specifically provisioned in their design. This, however, has little to do with the notion of autonomous agents.

In this chapter we further extend our agent architecture to enable agents to process norms and modify their behaviour so as to comply with these norms, should they choose to do so. In order to accomplish this, we extend a BDI language, allowing it to modify plans at runtime in reaction to newly accepted norms. Our main contribution is the introduction of plan manipulation strategies to enable reasoning about norms and to ensure compliance with new norms. Moreover, we provide new AgentSpeak(L) constructs using Jason [Bordini et al., 2005a] that allows our strategy to be implemented in AgentSpeak(L) agents.

The chapter is structured as follows: in Section 6.2 we outline our understanding of norms in agent languages; in Section 6.3 we describe our behaviour modification algorithms; in Section 6.6 we describe an AgentSpeak(L)-based implementation of our solution using a meta-level toolkit; in Section 6.7 we summarise related work relevant to these normative agents; and finally, in Section 6.8 we conclude the chapter.

6.2 Overview of Norm Processing

6.2.1 Norm Representation

In order to add normative processing to an agent language, it is necessary to provide some sort of representation for norms. Norms are often situated within the context of an electronic institution, making norms part of the environment [Vasconcelos et al., 2007]. Here, however, we are not concerned with the issues of electronic institutions or the modelling of a normative environment, but with the aspects that arise after norms have been perceived by an agent. We are concerned with the operational reasoning of agents, so our focus is on the consequences of norm compliance on an agent reasoning process. It is important to note that there are two major perspectives regarding normative systems: one with norms that, *by design*, cannot be violated, which is the perspective taken by electronic institutions [Garcia-Camino et al., 2005]; and another with norms that *can* be violated, potentially entailing penalties for the violator [Jones and Pörn, 1986]. The latter perspective has been gaining

Norm	Meaning
$obligation(p)$	add a goal to achieve state p
$obligation(a)$	add a new plan with an action a in its body.
$prohibition(p)$	prevent adoption of plans that bring about state p.
$prohibition(a)$	prevent adoption of plans that execute action a.

Table 6.1: Summary of norms and their meaning.

increasing acceptance within computer science [Faci et al., 2008; Oren et al., 2008a], since it considers that autonomous agents must always have choice over their behaviour. Since this thesis focuses on autonomous agents and how to achieve flexible behaviour, we take precisely this approach. Moreover, since we are concerned with agent *languages*, it is desirable to model norm acceptance in terms of agent language constructs, such as perception events. Therefore, we require a norm representation schema that eases the connection to agent languages.

From an individual agent's perspective, the main effect of norms on its reasoning is to determine which behaviours *must* be carried out, and which behaviours *must not* be carried out or else some form of punishment ensues. Of course, other types of norms prescribing preferences or more complex stipulations are possible, but from a practical perspective, the possibility of non-binding recommendations can be ignored, leaving an agent's own reasoning process to determine the best courses of action. We assume a closed world where everything not explicitly prohibited is permitted. This means that we ignore permissions, since they just recommend behaviours rather than *require* behaviours to be changed. Although we could include permissions as norm-modification operators in the sense that a permission may invalidate an obligation or prohibition, the resulting system would have effectively the same functionality in terms of behaviour modification, yet would complicate the discussion in this thesis.

6.2.2 Norms and Goal Types

Norms may refer to either: declarative world-states, in which an agent must try to achieve or refrain from achieving certain world-states; or actions, in which an agent must try to execute or refrain from executing a particular action [Norman and Reed, 2001], as is summarised in Table 6.1.

In our view, norms must have a well defined validity period; that is, a specification of when a certain norm is in force and when it ceases to be in force. This validity period is crucial for the enforcement of norms, particularly in the case of obligations, as without a deadline, an obliged party is not compelled to fulfill its obligation at any particular time. For example, if a consumer is obliged to pay his or her electricity bill, it is not possible to detect a violation without knowing when this payment is due. Conversely, for prohibitions, certain industrial research positions require a researcher to sign a form of non-disclosure agreement that includes provisions forbidding the researcher to engage in any competing research for a set period of time, usually one year after the termination of the initial contract. Therefore, norm encodings normally include a representation of the activation and expiration of each norm, indicating when the deontic modality referred to in the norm (*e.g.* obligation and prohibition) should be complied with.

Now, since most agent languages, such as AgentSpeak(L), 3APL, and others [Bordini et al., 2005a] use first-order logic to represent beliefs, it is appropriate to adopt a similar type of norm representation. Such a logic-based representation of norms allows for a more straightforward use of an agent language in detecting when norms are being complied with. Moreover, we leverage representational concepts from the formalisation of Oren *et al.* [Oren et al., 2008b], which includes notions of activation and expiration of a norm, delimiting their validity through time, an important aspect of norms in a dynamic environment. We therefore adopt a representation for norms in our system as follows:

$$norm(Activation, Expiration, Norm)$$

where *Activation* is the *activation condition* for the norm to become active, *Expiration* is the *expiration condition* to deactivate the norm, and $Norm$ is the norm itself. For example, if an agent is obliged to drive on the left when it is in Britain, but not when it leaves, such a norm is represented as `norm(in( britain ),notIn( britain ), obligation (driveOn( left )))`. In this context, we focus on using norms to determine whether plans and actions may be executed, and on the introduction of new triggering events linked to plans to satisfy new obligations.

It is important to note that we are not concerned, at this point, with handling more complex norm representation schemes in the way that a complete deontic logic does. Rather, we reduce norm representation to these two outcomes of prohibition and obligation to facilitate the creation of concrete agent behaviours aimed at complying with a set of norms. Ultimately, however, norms created with a more complex representation language must be reduced to these two outcomes in order to enable meaningful modifications to an agent's behaviour to be inferred.

There are many possible interpretations for such activation and expiration conditions, so it is important to specify what each condition means for an agent. We consider the activation condition to be a logical formula that, once entailed by an agent's beliefs, results in the norm being applicable to the agent. Conversely, we consider the expiration condition to be a logical formula that, once entailed by an agent's beliefs, results in the norm ceasing to be applicable to the agent. In what follows, we assume these conditions under a monotonic logic framework, so that when a condition becomes true, it will remain true in the future. We make this assumption now in order to simplify the description of our behaviour modification technique, so that norms can be activated and expire only once through their activation and expiration conditions. In this way, once a norm has been accepted and acted upon, there is no need for an agent to keep track of which norms have been accepted in the past, since monotonicity ensures their activation does not occur multiple times. We can later drop this assumption, but it is useful to keep it for ease of explanation, eliminating the need for more elaborate bookkeeping in our algorithms. If the monotonic assumption is dropped, our unmodified system can still cope with multiple instances of norms, but each activation can be considered in the context of a new norm addition, requiring the plans to be generated multiple times and resulting in a minor loss of computational efficiency. Alternatively, we can modify our algorithms to do the additional book keeping to cope with norms being reactivated, thus avoiding the need to generate the same plans multiple times for the same norm.

6.2.3 Norm Perception

In relation to *accepting* a norm, we consider its life cycle relative to an agent to start with the norm being issued in an environment (or society), and perceived by an agent. When an agent perceives new norms it decides whether or not to accept them (and modify its behaviour) [Garcia-Camino et al., 2005], or reject them (and suffer potential sanctions). Our view contrasts with that taken by *electronic institutions*, in which norms cannot be rejected and are complied with by design. Although the issue of perceiving and deciding on accepting a norm has seldom been considered in the literature, it is of considerable importance in open dynamic systems regulated by norms. For example, in real human societies, norms are not *inherent* to the world, but rather created by some relevant authority and then made known to the parties to which a norm applies under the expectation that these norms will be accepted and complied with, which is not always the case. Furthermore, when an agent enters a new environment, it is possible that it will encounter a set of norms different from those for which it was designed. Here, an agent must be made aware of these norms and change its behaviour accordingly.

We use the distinction of perception and belief to denote whether or not a norm has been accepted. If an agent has added a perceived norm to its belief-base, we consider the agent to have implicitly accepted the norm, and behaviour modification must ensue. In order to facilitate the integration of deontic states into an agent system, we adopt the view of Pörn [Pörn, 1970] that deontic statements themselves can be seen as states of affairs. Thus, from an agent's perception point of view, the fact that a book is over a table is no different than a norm obliging an agent to drive on the right. Given this representation, the difference between perception and belief indicates a norm's status in terms of acceptance by an agent.

After accepting a norm, an agent needs to make sure that the norm is not in conflict with its set of norms already in effect and then modify its behaviour. For example, if an agent accepts a norm n_1 that prohibits it from performing an action *act*, and later perceives a norm n_2 that obliges it to perform action *act*, then n_1 and n_2 are in conflict. Thus, it is necessary for an agent to make sure it does not accept conflicting norms which would result in inconsistent behaviours, such as creating and executing plans that carry out prohibited actions.

These processes are illustrated in Figure 6.1, in which rectangles represent agent reasoning processes, rounded rectangles represent environmental or societal processes, and diamonds represent decisions within an agent. The flow starts with the receipt of norms from the environment and the decision to accept or reject a norm. The processes that follow from this decision (including potential sanctions) consist of verifying consistency and subsequently changing behaviour. In this thesis, we focus only on the behaviour modification part at the end of the flow in the figure, and indicated in a dashed box. In particular, we avoid the issue of maintaining norm consistency, a very complex problem in itself, having been addressed by other efforts [Vasconcelos et al., 2007] that can be easily integrated into ours, as we discuss in our conclusions.

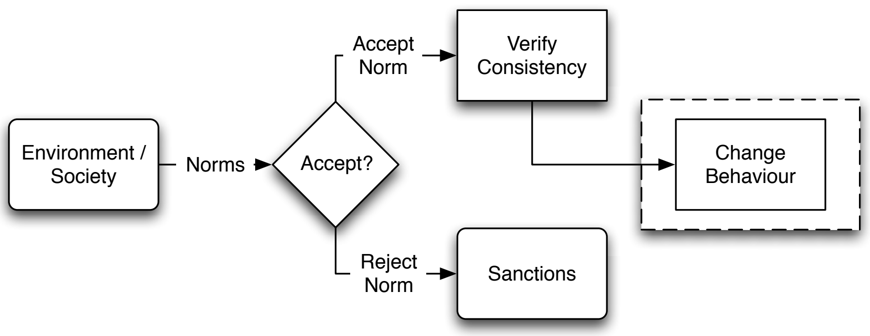


Figure 6.1: Overview of norm processing in our system.

The point here is that while norms have been considered more generally, especially in the context of societies, the impact of norms on agent architectures, in particular BDI architectures, has received little attention. Specifically, the impact of norms at the level of an agent's control cycle has largely been overlooked. Therefore, building on previous efforts, in particular the planning capabilities described in Chapter 3, and the communication framework explained in Chapter 5, we address norms from the point at which an agent receives them.

In the following sections we proceed to detail the expected actions an agent must undertake in order to comply with norms under various combinations of activation and expiration conditions.

6.3 Norm Outcomes

As we have seen, our norm representation includes conditions for norm activation and expiration, denoting when these norms are to affect an agent's behaviour. The main consequence of these conditions when modifying behaviour lies in what must be done in reaction to the agent accepting new norms. The two most evident situations regarding activation and expiration conditions are: when the activation condition is already true, which means that the norm must be enacted immediately and any contrary behaviours stopped; and when an expiration condition is already true, which means that the norm has expired and can be ignored. Clearly, there are a number of possible combinations of activation and expiration conditions, and each combination results in a different set of actions an agent must carry out in order to comply with the norm. We therefore analyse these combinations providing an overview of the outcome of each combination for an agent wanting to comply with the norm. Subsequently, we detail algorithms that generate these outcomes, starting with algorithms to react to the activation of a norm in Section 6.4, followed by the results of norm expiration in Section 6.5.

We summarise all norm condition combinations and their outcomes in Table 6.2, for each type of deontic modality considered in this thesis (*i.e.* obligation and prohibition), and for each type of target for the modality (either an action or a world-state). The activation and expiration condition columns give the truth-value of these conditions at the time the agent accepts the norm, so that (following our monotonic assumption) *True* means the expression is true and will remain so, while *False* means the expression is not yet true.

The first case we consider for all deontic modalities is when both the activation and the expiration condition are true, which results in the norm being ignored, as its expiration condition has already elapsed. Norms are also ignored when the activation condition is false but the expiration condition is true since, again, the norm has already expired. Now, if the activation condition is already true when an agent accepts an obligation, it must react to either achieve the world-state specified in an $obligation(p)$, or execute the action specified in an $obligation(a)$.

If this same combination of conditions occurs for a prohibition, it means that an agent must refrain from achieving the offending world-state or executing the offending action. Since an agent might have already adopted an offending plan, it must not only suppress plans that might violate the prohibition, but also drop any instances of these plans adopted as intentions.

Finally, when both the activation and expiration conditions are false, an agent must create new plans (using the planning capabilities introduced with AgentSpeak(PL) in Chapter 3) to enforce compliance as soon as the activation condition holds and add them to the plan library. This plan creation step is necessary because in this situation the norm has not yet been activated, but the agent must be prepared for its activation in the *future*. In particular, if the norm refers to an obligation to achieve a world-state, the agent must create a plan that achieves the specified world-state whenever the activation condition holds, whereas if it refers to an obligation to execute an action, the new plan must execute that action. Note that plans to comply with obligations to achieve world-states are created using some kind of

Deontic Modality	**Activation Condition**	**Expiration Condition**	**Outcome**
$obligation(p)$	True	True	Ignore norm
	True	False	Adopt plan to achieve p
	False	False	Add plan to achieve p to the plan library whenever the activation condition holds
	False	True	Ignore norm
$obligation(a)$	True	True	Ignore norm
	True	False	Adopt plan to that includes a
	False	False	Add plan that includes a to the plan library whenever the activation condition holds
	False	True	Ignore norm
$prohibition(p)$	True	True	Ignore norm
	True	False	Drop intentions to achieve p and suppress plans that achieve p
	False	False	Add plan to suppress plans that achieve p whenever the activation condition holds
	False	True	Ignore norm
$prohibition(a)$	True	True	Ignore norm
	True	False	Drop intentions that include a and suppress plans that include a
	False	False	Add plan to suppress plans that include a whenever the activation condition holds
	False	True	Ignore norm

Table 6.2: Combinations of activation and expiration conditions and their outcomes.

planning capability, of which there are many, though here we adopt that of AgentSpeak(PL) in Chapter 3. Furthermore, if the norm refers to a prohibition to achieve a world-state, the new plan must suppress all plans that achieve the prohibited world-state whenever the activation condition holds, whereas if it refers to a prohibition to execute an action, the new plan must suppress all plans containing the prohibited action. In both these cases, not only must plans be suppressed, but violating intentions must also be dropped.

Now, we understand that creating new plans to comply with newly accepted norms, and later removing these plans after a norm has expired, from a pragmatic point of view, is not necessarily the optimal solution. For example, it is certainly possible for an agent to use existing plans in its plan library to accomplish an obligation, avoiding the need to generate a new plan from scratch. However, an agent must still ensure that the plan's invocation condition is compatible with the activation condition of the obligation, as well as ensuring that it does not also bring about undesirable effects. This analysis process is complex in itself, and it is unclear how the cost of this process compares to planning from first principles. Furthermore, removing plans makes perfect sense under our monotonic assumption, since a norm will never occur in the exact same form again in the future. However, if we drop the monotonic assumption, plans should no longer simply be deleted after the norm has expired, but must instead be suppressed and stored for reuse for norms with multiply recurring activation and expiration conditions. Therefore, the dropping of the monotonic assumption does not invalidate our algorithms, but rather requires some additional bookkeeping.

6.4 Norm Activation

Now that we have considered the possible high-level outcomes for an agent seeking to comply with norms, we consider how to deal with the receipt of obligations and prohibitions in more detail. In what follows, we assume only a BDI-type language with constructs for *goals to be* or *goals to do* or both; and a plan library or similar construct to store plans, which can be modified to reflect the set of possible plans an agent may adopt. This plan library is the target of our norm processing mechanism, which generates changes in the set of possible plans an agent may adopt either by adding new plans to comply with obligations or by preventing existing plans from being executed to comply with prohibitions. To accomplish these changes, we need to modify an agent's reasoning ability to be able not only to deal with the world through its actions, but also to deal with its own data structures, its available plans in particular. As a consequence, we need meta-level operators that can suppress plans from being selected from a plan library or from being generated by a planner, and can introduce new plans to the plan library.

In the following sections we refer to some algorithms as *templates*, or abstract *algorithms* that need to be further specified at runtime in order to be instantiated as concrete *algorithms*. This is because most agent languages define agent behaviours in terms of reactions to certain conditions in the environment, and the norm processing behaviours we specify are defined in terms of their activation and expiration conditions. More specifically, we consider the addition of plans to handle activation conditions for each type of deontic modality in the algorithms shown in Sections 6.4.1 and 6.4.2, and the plans to handle expiration conditions in Section 6.5.

Now, in order to illustrate the operation of these algorithms, we use an AgentSpeak(L) implementation as an example, instantiating the algorithms appropriately. For this to be meaningful, we must recap on the notation used for an AgentSpeak(L) plan $e : b_1 \& \ldots \& b_m \leftarrow h_1; \ldots; h_n.$, which contains a triggering event e, a context condition expressed by the $b_1, \ldots, b_m$ belief literals, and a list $h_1, \ldots, h_n$ of goals or actions [Rao, 1996]. In addition, we need to differentiate actions (which can also include *internal actions* representing some functionality internal to an agent) from beliefs, which we do with a preceding dot symbol; then, the action to vacuum a room is represented as `.vacuum(room)`, and the action to remove a plan from the plan library is written as `.remove_plan(Plan)`, whereas the belief that a room is clean is represented as `clean(room)`. (Note that the specifics of how particular internal actions work are not important at this point, and we discuss them in detail later in Section 6.6 along with other implementation concerns.) Finally, we adopt the plan labelling convention used in the Jason interpreter [Bordini et al., 2007], whereby plans are given unique labels in the form of predicates preceded by the *at* (@) sign. For example, a plan `@id e : c <- a.` is uniquely identified by the label `id`, which can be used later for plan manipulation operations referring to the plan `e : c <- a.`.

6.4.1 Activating Obligations

When an agent chooses to comply with a newly perceived obligation, the steps it must take are as stated in more detail in Algorithm 5. While this is an algorithm, it is also a plan, since the algorithm must be encoded as a plan for an agent to be able to use it. The plan is very simple and follows the basic requirements of the outcomes of accepting an obligation as shown in Table 6.2. That is, if an obligation has already expired, it can be ignored; otherwise, new plans must be added to handle the activation and expiration conditions of the norm. If the obligation has been activated, it must be acted upon immediately. Thus, Line 1 checks whether the agent believes the norm has expired, and if so, ignores it. Next in Line 4, the agent adds a new plan, from Algorithm 6, to deal with the activation condition. If the norm activation condition is true, this plan must be performed immediately, either achieving a specified world-state or executing an action. Finally, the agent must add another new plan to handle the expiration of the obligation, which we explain in Section 6.5. Since we are not concerned in this thesis with the process by which an agent decides whether to comply with a norm, in both plans we assume that *acceptance* has been determined, indicated in the pre-requisites of each algorithm. In Algorithm 5, the requirements also refer to an agent's belief base, used to specify that the activation condition must hold before seeking to comply.

Algorithm 5 Plan to comply with an obligation.

```
Require: Receipt of $norm(Activ, Exp, obligation(O))$
Require: Acceptance of $norm(Activ, Exp, obligation(O))$
Require: Belief Base $BF$
Require: Plan Library $PL$
1: if $BF \models Exp$ then
2:     return
3: end if
4: Create plan $L_{Activ,obligation(O)}$ using Algorithm 6 template
5: Add $L_{Activ,obligation(O)}$ to $PL$
6: if $BF \models Activ$ then
7:     Execute plan from Algorithm 6
8: end if
9: Add plan from Algorithm 9 to deal with expiration
```

The newly added plan to deal with the activation condition is detailed in Algorithm 6, and consists of either adding a new goal to achieve an obligatory world-state, or executing an obligatory action upon the perception of the norm activation event. Note that Algorithm 6 is a plan *template*; that is, it shows the general form of plans that are instantiated with information about a specific norm and its associated activation condition.

Algorithm 6 Plan template to react to activation of an obligation.

```
Require: Acceptance of $norm(Activ, Exp, obligation(O))$
Require: Receipt of $Activ$ event
Require: Plan is uniquely labelled with label $L_{Activ,obligation(O)}$
1: if $O$ is a world-state $p$ then
2:    Add goal to achieve $p$
3: else if $O$ is an action $a$ then
4:    Add goal to execute $a$
5: end if
```

Example To illustrate these algorithms (plans), we use the example of a cleaner agent that is capable of using a vacuum cleaner to clean a room. Suppose that the cleaner agent accepts a norm to achieve a world-state in which the floor is clean at *8:00* hours everyday, until Christmas Day, expressed as +norm(**time**(800), day(xmas), obligation(clean(floor)))[source(env)]. In our solution, this results in the generation of two plans, one associated with the activation condition, which leads to the adoption of a plan to achieve the obliged world-state, and one associated with the expiration condition, which leads to the removal of all the plans associated with the specified norm. The two plans, using our notation introduced earlier, are shown in Table 6.3 (these plans contain implementation- specific detail, for which we delay explanation until Section 6.6). The first plan, labelled obligationStart (clean(floor)), has the activation condition **time**(800) as its trigger, which leads to the adoption of an achievement goal !clean (floor), (and in turn a plan) assumed to achieve a state in which clean(floor) holds. This goal addition, shown in Line 3 of Table 6.3, corresponds to Line 2 of the generic Algorithm 6. The second plan, labelled obligationEnd(clean(floor)) has the expiration condition day(xmas) as its trigger, and results in both plans being removed from an agent's plan library through two invocations of the .remove_plan internal action. Note that the steps to remove the plans associated with the obligation's expiration may seem to be self-referential but, due to the AgentSpeak (BDI) reasoning cycle, on execution of the plan of Table 6.3 these steps are loaded into an instantiated intention that removes the original plans from the plan library.

```
1 @obligationStart(clean(floor))
2 +time(800) : true
3  <- !clean(floor).
4
5 @obligationEnd(clean(floor))
6 +day(xmas) : true
7  <- .remove_plan(obligationStart(clean(floor)));
8      .remove_plan(obligationEnd(clean(floor))).
```

TABLE 6.3: Plans for the clean state norm.

```
@obligationStart(vacuum(floor))
+time(800) : true
 <- .vacuum(floor).

@obligationEnd(vacuum(floor))
+day(xmas) : true
 <- .remove_plan(obligationStart(vacuum(floor)));
     .remove_plan(obligationEnd(vacuum(floor))).
```

TABLE 6.4: Plans for the vacuum action norm.

Conversely, when an agent is obliged to execute an action, a similar plan schema is generated, but instead of adopting a goal to achieve a specified state, the new plan needs simply to execute the specified action. Thus, if a cleaner accepts a norm to vacuum the floor at 8:00 hours every day, until Christmas day, expressed as +norm(**time**(800), day(xmas), obligation(.vacuum(floor)))[source(env)], very similar plans are adopted, but instead of adding an achievement goal to its intention structure, the specified action is executed, as illustrated in the plans of Table 6.4.

6.4.2 Activating Prohibitions

Now, when the accepted norm refers to a prohibition, we need to use Algorithm 7. This consists of first checking for norm expiration, then adding a plan to handle norm activation, and checking whether or not the norm has already been activated, in which case the added plan must be executed.

Algorithm 7 Plan to comply with a prohibition.

```
Require: Receipt of $norm(Activ, Exp, prohibition(P))$
Require: Acceptance of $norm(Activ, Exp, prohibition(P))$
Require: Belief base $BF$
1: if $BF \models Exp$ then
2:     return
3: end if
4: Create plan $L_{Activ,prohibition(P)}$ using Algorithm 8 template
5: Add $L_{Activ,prohibition(P)}$ to $PL$
6: if $BF \models Activ$ then
7:     Execute plan from Algorithm 8
8: end if
9: Add plan from Algorithm 10 to deal with expiration
```

The plan to deal with prohibition activation, detailed in Algorithm 8, consists of first scanning an agent's intentions (*i.e.* the plans already adopted) for instances of the prohibited action, or for plans having a prohibited world-state as a consequence. If plans violating the prohibition are found as intentions, they must be dropped immediately. Afterwards, the plan library is similarly scanned for plans that may violate the prohibition if executed. If the prohibition refers to a state, all plans with this state as a consequence must be suppressed, while if the prohibition refers to an action, all plans with this action must be suppressed. In addition to suppressing the plans, suppressed plans are stored in the set $S_{Plans,prohibition(P)}$, so that when the prohibition expires later, these plans can be restored. Like in Algorithm 6, the plan of Algorithm 8 is a plan *template* (that is, a general form of plan that becomes concrete after the information about a specific norm and its activation and expiration conditions is known).

Algorithm 8 Plan template to react to state prohibition activation.

```
Require: Acceptance of $norm(Activ, Exp, prohibition(P))$
Require: Receipt of $Activ$ event
Require: Intention structure $I$
Require: Plan library $PL$
Require: Plan uniquely labelled with label $L_{Activ,prohibition(P)}$
Ensure: Suppressed plans are stored in set $S_{Plans,prohibition(P)}$
 1: for all Intention $i \in I$ do
 2:   if ($P$ is a world-state $p$) and ($p$ is a consequence of $i$) then
 3:     Drop intention $i$
 4:   else if $P$ is an action $a$ then
 5:     for all Steps $s$ in remaining steps of $i$ do
 6:       if $s = a$ then
 7:         Drop intention $i$
 8:       end if
 9:     end for
10:   end if
11: end for
12: for all Plans $pl \in PL$ do
13:   if ($P$ is a world-state $p$) and ($p$ is a consequence of $i$) then
14:     Suppress $pl$
15:     $S_{Plans,prohibition(P)} = S_{Plans} \cup pl$
16:   else if $P$ is an action $a$ then
17:     for all Steps $s$ in $pl$ do
18:       if $s = a$ then
19:         Suppress $pl$
20:         $S_{Plans,prohibition(P)} = S_{Plans} \cup pl$
21:       end if
22:     end for
23:   end if
24: end for
```

It is important to note here that we leave the meaning of plan suppression relatively ambiguous, since the details of how this is done depend on the particular way in which an agent architecture operates. Later in this chapter we show how this can be achieved in an AgentSpeak(L)-type agent.

Example If our example cleaning agent was prohibited from entering a room with classified documentation in it, expressed as `+norm(`**`time`**`(800), day(xmas), prohibition(in(classifRoom)))[source(env)]`, two new plans need to be generated. First, for activation, we need to make sure that the plans that result in the cleaner being in the classified room are *suppressed* from execution, as shown in Table 6.5. In this plan, Lines 3 to 5 of Table 6.5 correspond to Lines 13 to 15 of the generic Algorithm 8. Moreover, when the prohibition expiration condition becomes true, not only do the plans to handle the activation and expiration conditions need to be removed, but also the plans that were suppressed by the activation condition need to be *unsuppressed*.

```
1   @prohibitionStart(in(classifRoom))
2   +!Start : true
3    <- !findPlansWithEffect(in(classifRoom), SPlans);
4       !suppressPlans(SPlans);
5       +suppressedPlans(in(classifRoom),SPlans).
6
7   @prohibitionEnd(in(classifRoom))
8   +!End : suppressedPlans(in(classifRoom),SPlans)
9    <- !unsuppressPlans(SPlans);
10      -suppressedPlans(in(classifRoom),SPlans);
11      .remove_plan(prohibitionStart(in(classifRoom)));
12      .remove_plan(prohibitionEnd(in(classifRoom))).
```

Table 6.5: Plans generated from a state prohibition.

Plans to effect restrictions on executing actions are very similar to those relating to achieving world-states, the only difference being in the process for selecting the plans that need to be suppressed. In this case, the plans searched for are those that contain a particular action. For example, if the cleaning agent is obliged not to vacuum a table during its rounds of cleaning through the norm +norm(**time**(800), day(xmas), prohibition(vacuum(table)))[source(env)], the plans to effect this prohibition are those shown in Table 6.6.

```
@prohibitionStart(vacuum(table))
+!Start : true
 <- !findPlansWithAction(vacuum(table), SPlans);
    !suppressPlans(SPlans);
    +suppressedPlans(vacuum(table),SPlans).

@prohibitionEnd(vacuum(table))
+!End : suppressedPlans(vacuum(table),SPlans)
 <- !unsuppressPlans(SPlans);
    -suppressedPlans(vacuum(table),SPlans);
    .remove_plan(prohibitionStart(vacuum(table)));
    .remove_plan(prohibitionEnd(vacuum(table))).
```

Table 6.6: Plans generated from an action prohibition.

It is important to note at this point that our strategy does not address the issue of norm consistency; that is, we assume that whenever plans are added to handle a certain norm, the norm is consistent with both the agent's objectives and other norms previously accepted by the agent. It is, however, possible to extend our work with a norm consistency maintenance strategy such as that proposed by Vasconcelos *et al.* [Vasconcelos et al., 2007], but such issues are outside the scope of this thesis.

6.5 Norm expiration

Now that we have seen the plans needed to start complying with norms under several circumstances, we need to examine how agent behaviour is modified as a result of a norm expiring. When an agent accepts a norm and changes its behaviour as a result of the norm becoming active, it either includes extra plans to comply with obligations or suppresses some of its plans in order not to violate a prohibition. However, these behaviour modifications should not become permanent within an agent if the norms that caused them cease to be active. Moreover, our monotonicity assumption entails that once a norm has been activated and then expired, it will never become active again. Thus, Algorithms 5 and 7, containing plans for reacting to norms, also include a final step to add a plan dealing with norm expiration to the plan library. Such norm expiration plans aim to *undo* the behavioural changes effected when the norms were activated, thus restoring the plan library to a state in which an agent's behaviour is not affected by them. Thus, the plan in Algorithm 9 consists both of removing the plan responsible for dealing with obligation activation and afterwards of removing itself from an agent's plan library. Both these plans must be individually identifiable within an agent's plan library, so we label them respectively $L_{Activ,obligation(O)}$ and $L_{Exp,obligation(O)}$ in order to remove them when they are no longer needed.

Algorithm 9 Plan to react to the expiration of an obligation.

```
Require: Acceptance of $norm(Activ, Exp, obligation(O))$
Require: Receipt of $Exp$ event
Require: Label $L_{Activ,obligation(O)}$ for a norm activation plan
Require: Plan library $PL$
Ensure: Plan is uniquely labelled with label $L_{Exp,obligation(O)}$
  1: Remove plan $L_{Activ,obligation(O)}$ from $PL$
  2: Remove plan $L_{Exp,obligation(O)}$ from $PL$
```

The acceptance of prohibitions, on the other hand, not only adds new plans to react to norm activation and expiration, it also affects which plans are available to an agent after a prohibition has been activated. Thus, the plan to react to the expiration of a prohibition must not only remove the new plans added to comply with the norm, it must also restore the plans previously suppressed to their initial state of availability. The plan of Algorithm 10 accomplishes this by unsuppressing the initially suppressed plans which, according to the plan of Algorithm 8, were stored in the set $S_{Plans,prohibition(P)}$, and then removing plans $L_{Activ,prohibition(P)}$ and $L_{Exp,prohibition(P)}$ from the plan library.

6.6 Normative AgentSpeak(L)

In order to test the viability of our solution in a practical agent language, we have developed an implementation of the strategies outlined in Section 6.3 using an AgentSpeak(L)

```
Algorithm 10 Plan to react to the expiration of a prohibition
Require: Acceptance of $norm(Activ, Exp, prohibition(P))$
Require: Receipt of $Exp$ event
Require: Label $L_{Activ,prohibition(P)}$ for a norm activation plan
Require: Plan library $PL$
Require: $S_{Plans,prohibition(P)}$ of suppressed plans
Ensure: Plan is uniquely labelled with label $L_{Exp,prohibition(P)}$
 1: Unsuppress all plans from $S_{Plans,prohibition(P)}$
 2: Remove plan $L_{Activ,prohibition(P)}$ from $PL$
 3: Remove plan $L_{Exp,prohibition(P)}$ from $PL$
```

interpreter. An important part of this involves the manipulation of an agent's own plan library, necessitating a means to perform meta-level operations, allowing AgentSpeak(L) plans to manipulate other plans. With such a meta-level facility in place, we can create AgentSpeak(L) plans that accomplish the norm-induced behaviour modification described above.

6.6.1 Norm acceptance in AgentSpeak(L)

As we have seen in Section 6.2.3, we consider that an agent's perception of a norm is no different to the perception of any other facts about the world. As a consequence, it is possible to model norm perception in AgentSpeak(L) in exactly the same way as environmental perception, which in turn allows us to associate AgentSpeak(L) plans with the events connected to new environment percepts. Here, when a norm is perceived, an agent receives an event +norm(Act,Exp,obligation(O))[source(env)], which is handled by an AgentSpeak(L) plan containing the procedure to decide whether or not to accept the norm. The annotation mechanism serves to show who or what issued the norm, so that environmental norms (*i.e.* norms inherent to a certain environment) that are not issued by any agent are different from norms that are issued by other agents. The difference in origins of norms can then be taken into account when deciding whether to comply with them. Norms coming from from agents may be, for example, the result of signing a contract between them. When events in the form norm(Activation, Expiration, Norm) are posted to an agent it must decide whether or not to accept the new norm. The decision to accept or reject these newly discovered norms can then be associated with plans having such events as their triggering conditions, as illustrated in Table 6.7, which consists of first accomplishing a test goal querying whether or not the norm should be accepted, and then accomplishing an achievement goal to process the norm.

```
+norm(Activation, Expiration, Norm)[source(Src)] : true
 <- ?acceptNorm(Activation, Expiration, Norm, Src);
    !processNorm(Activation, Expiration, Norm).
```

TABLE 6.7: Plan for norm receipt.

It is important to note that we do not detail how the decision to accept norms is made; for our purposes, it suffices to provide a plan (in Table 6.8) that abstracts this procedure, by accepting all norms unconditionally, but could easily include some stronger analysis of the benefit of accepting a norm versus its associated penalties.

```
?acceptNorm(Activation, Expiration, Norm, Src) : true
 <- true.
```

TABLE 6.8: Plan for norm acceptance.

With this norm representation, we can now describe the machinery necessary to process newly acquired norms and modify an agent's behaviour. As we have seen, norms can either refer to the imposition of new behaviours or the suppression of existing ones, and norms have activation conditions indicating when they become effective, as well as expiration conditions indicating when they cease to be effective. From an AgentSpeak(L) perspective, this coincides with the notion of triggering conditions in plans, thus requiring that new plans be created to respond to both the activation condition and the expiration condition of a norm, as illustrated in Figure 6.2.

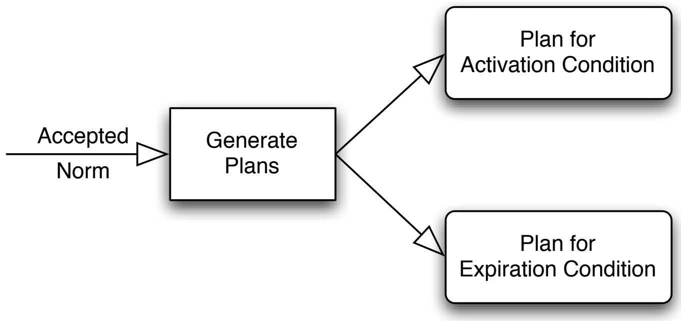


FIGURE 6.2: Effects of norm acceptance.

6.6.2 Meta-level actions for AgentSpeak(L)

The AgentSpeak(L) language does not have explicit constructs for the analysis of a plan library, yet this is required in the strategies described in Section 6.3 and implemented in Section 6.6.3. In particular, for an agent to evaluate its existing behaviours, encoded in its plans, we require the introduction of meta-level operators that allow regular AgentSpeak(L) plans themselves to explore and process other plans in the plan library. In our system, we construct such operators, as summarised in Table 6.9, through the use of *internal agent actions*. We have seen in Section 5.3.1 that internal actions are non-world changing operators that can be executed instantaneously, and allow the construction of custom procedures within AgentSpeak(L). We recap that internal actions are denoted by a preceding dot, so the internal action to suppress a plan is represented as `.suppress_plan(Plan)`.

Action	Effect
`.plan_steps(P,S)`	takes a plan P and unifies its plan steps as a list of literals with S
`.plan_consequences(P,C)`	takes a plan P and unifies its declarative consequences with C
`.action(A)`	succeeds if the A atom refers to an action
`.literal(L)`	succeeds if the L atom refers to a literal
`.add_plan(P)`	adds P to the plan library
`.remove_plan(P)`	removes P from the plan library
`.suppress_plan(P)`	suppresses the specified plan from being executed
`.unsuppress_plan(P)`	allows a previously suppressed plan to be executed

Table 6.9: Summary of the meta-level actions

Most of the meta-level actions of Table 6.9 either have simple outcomes or implement parts of the algorithms described in Section 6.3. However, the first two actions in the table are needed specifically to deal with the way in which AgentSpeak(L) operates, and we need to clarify them further. The action .plan_steps(P,S) takes an AgentSpeak(L) plan in variable P and unifies the steps of this plan with a list in variable S, while .plan_consequences(P,C) takes a plan in variable P and unifies the declarative consequences with a list in variable C. This latter internal action works similarly to the process of extracting declarative information used in AgentSpeak(PL) in Section 3.5.1. The next two actions in Table 6.9 are used in the context conditions of the plans to deal with actions and literals, identifying whether the target atom in a norm refers to an action or a belief literal. Finally, since we need to manipulate plans in the plan library, we use action .add_plan(P) to add a plan P to the plan library and .remove_plan(P) to remove the plan specified in P, while .suppress_plan(P) and .unsuppress_plan(P) are used respectively to suppress (and therefore prevent from executing) a plan, and to unsuppress a plan (reversing any possible suppression).

Plans to ensure compliance with prohibitions are more complex in that they require an agent to scan its entire plan library looking for violating plans. For prohibitions relating to executing an action, this requires finding all plans in the plan library that contain the prohibited action and suppressing their execution. This is shown in Algorithm 11.

Prohibitions relating to achieving certain world-states require an agent to analyse the effects of each of the plans in its plan library, and suppress the execution of those that have the prohibited state as an effect. An algorithm to accomplish this is shown in Algorithm 12.

Algorithm 11 Find plans with action.

```
Require: AgentSpeak plan library $PL$
Require: Action $act$
Ensure: A list $PL_A$ of plans containing $act$
1: for all Plans $\{t : c \leftarrow b.\} \in PL$ do
2:   for all Steps $s_i \in b$ do
3:     if $s_i$ unifies with $act$ then
4:       Add $\{t : c \leftarrow b.\}$ to $PL_A$
5:     end if
6:   end for
7: end for
8: return $PL_A$
```

Algorithm 12 Find plans with effect.

```
Require: AgentSpeak plan library $PL$
Require: Proposition $p$
Ensure: A list $PL_P$ of plans containing $p$
1: for all Plans $\{t : c \leftarrow b.\} \in PL$ do
2:   Get the effects $E$ of $\{t : c \leftarrow b.\}$
3:   for all Effects $e \in E$ do
4:     if $e$ unifies with $p$ then
5:       Add $\{t : c \leftarrow b.\}$ to $PL_P$
6:     end if
7:   end for
8: end for
9: return $PL_P$
```

6.6.3 AgentSpeak plan modification mechanisms

Using these internal actions, we can create AgentSpeak(L) plans that add newly accepted norms, as described in Section 6.2.3, to the set of active norms. As we have seen, adding obligations is relatively straightforward, and we omit the meta-level plans for their addition, focussing instead on those responsible for handling the addition of prohibitions. When a prohibition referring to an action is added, we need to create two plans, one to handle the start of the prohibition and another to handle the end of the prohibition.

Thus, when a prohibition on an action becomes effective, an agent needs first to find all the plans with the prohibited action and then suppress each of the plans containing the offending action. Finding these plans involves a `!findPlansWithAction( Prohibition, SelPlans)` plan, which uses the `.plan_steps(Plan,Steps)` action to explore each plan step, corresponding to step 2 of Algorithm 11. Once the plans are identified, the plan needs to suppress these plans with the `!suppressPlans(SelPlans)` plan, which uses the `.suppress_plan(Plan)` action. In our implementation, plans that are suppressed are removed from the plan library, and thus not considered as options to achieve a certain goal. It is also important to keep track of the suppressed plans so that they can be unsuppressed later. When the prohibition ceases to be effective, we need to unsuppress the plans previously suppressed, as well as remove the plans related to this particular norm, since they will no longer be necessary. A meta-plan responsible for creating these norm-related plans uses the start and end conditions to create the triggers for two plans that accomplish the suppression and unsuppression of the necessary plans, generating plans following the template shown in Table 6.10.

```
@prohibitionStart(Prohibition)
+!Start : true
 <- !findPlansWithAction(Prohibition, SPlans);
    !suppressPlans(SPlans);
    +suppressedPlans(Prohibition,SPlans).

@prohibitionEnd(Prohibition)
+!End : suppressedPlans(Prohibition,SPlans)
 <- !unsuppressPlans(SPlans);
    -suppressedPlans(Prohibition,SPlans);
    .remove_plan(prohibitionStart(Prohibition));
    .remove_plan(prohibitionEnd(Prohibition)).
```

Table 6.10: Template plans generated from an action prohibition.

To add the plans that handle prohibitions on world-states, the necessary steps are similar to those for prohibitions on actions, with the only difference being that the search criterion for offending plans involves the effects of these plans. We extract the effects of plans using the `.plan_consequences(Plan, Consequences)` action, used in the `!findPlansWithEffect( Prohibition, SelPlans)` plan. Like in the prohibition for actions, a meta-plan responsible for creating these norm-related plans creates two plans following the template shown in Table 6.11.

```
@prohibitionStart(Prohibition)
+!Start : true
 <- !findPlansWithEffect(Prohibition, SPlans);
    !suppressPlans(SPlans);
    +suppressedPlans(Prohibition,SPlans).

@prohibitionEnd(Obligation)
+!End : suppressedPlans(Prohibition,SPlans)
 <- !unsuppressPlans(SPlans);
    .remove_plan(prohibitionStart(Prohibition));
    .remove_plan(prohibitionEnd(Prohibition)).
```

Table 6.11: Template plans generated from a state prohibition.

6.7 Related work

In this chapter, we have taken the basic notions of norms within agent systems and have used them to define individual agent behaviour modification mechanisms to comply with them, thus filling a gap in normative agent research. Previous work has also addressed similar concerns, in moving from largely intractable deontic modalities into simpler, yet useful representations of norms to be used in a concrete system. For example, one of the first practical architectures for a norm-driven agent was Kollingbaum and Norman's NoA [Kollingbaum and Norman, 2003], which takes a BDI-like agent architecture and changes the focus of agent behaviour from achieving desires to fulfilling norms. As in NoA, we use an explicit representation of the effects of an agent's plans to detect potential norm violations, as well as deciding which plans are more suitable for achieving an obligation, but our agents are still driven by their desires like traditional BDI agents. In contrast, Vazquez-Salceda *et al.* [Vázquez-Salceda et al., 2005] take the ISLANDER [Esteva et al., 2001] formalism and use it to establish guidelines for the implementation of a normative system, its monitoring and the enforcement of its norms. Our work can be seen as complementary, since we provide the machinery to modify the behaviour of an agent willing to accept norms, as opposed to being concerned with the rest of the system. Unlike other approaches, such as electronic institutions [Garcia-Camino et al., 2005], which typically *require* compliance, our agents may choose to ignore a norm, even if it may lead to potential penalties.

Due to the rather large scope of normative agents, we have explicitly not considered the important issue of maintaining the consistency of a set of norms. Other types of potential clashes involve overlapping of norm conditions, including their activation and expiration. For example, if an agent accepts a norm prohibiting work from time 12 to time 14, and another prohibiting work from time 11 to time 15, plans may be modified due to the activation of the second prohibition at time 11, and then modified again due to the expiration of the first prohibition at time 14, jeopardising the second prohibition in the process. However, it is important to point out that a solution for addressing this could be easily adapted from the work of Vasconcelos *et al.* [Vasconcelos et al., 2007], which provides an algorithm for resolving conflicts and inconsistencies in sets of norms using a unification-based technique. Similarly, our agents could resolve norm conflicts by adapting the mechanism used by Kollingbaum's [Kollingbaum and Norman, 2003] NoA architecture for the same purpose. Thus, by assuming an existing norm consistency maintenance process, we have reduced the problem we address, to focus on individual norm additions and deletions, avoiding the dilution of our efforts, and facilitating a more specific consideration of the relevant issues.

6.8 Conclusions

In this chapter, we have described a *framework* of concrete behaviours for classical agent languages that enable them to effect changes in their own plan libraries to conform to new norms accepted from the environment. Our framework is sufficiently generic that it can be extended into any traditional BDI style agent language. Importantly, we have also developed these general algorithms further into a concrete instantiation in AgentSpeak(L) (using a new toolkit of meta-level operators, that has not been considered previously), providing a novel contribution in itself, as well as an illustration and realisation of the algorithms. We show

how our framework can generate new plans to enable agents to comply with norms, and remove the plans when the norms are no longer relevant, through a series of examples throughout the chapter, demonstrating the practicality of our approach.

Regarding the evaluation of our normative processing technique, it is difficult to evaluate in a quantitative empirical analysis, and we have therefore demonstrated it through examples of the effects of the mechanism on concrete AgentSpeak(L) agents. Further analysis of the power and limits of our mechanism, however, is an ongoing process, and can only be achieved as more agents and normative systems are implemented using our norm processing techniques. This can and should be undertaken through extensive deployment and use, with experience feeding into evaluation and subsequent refinement. As for Chapter 5, while this is beyond the scope of what is possible in this thesis, it offers opportunities for further research.

The norm processing mechanism described in this chapter has been fully implemented using the Jason interpreter extended with the AgentSpeak(PL) planning capabilities[1]. Agents built using this system include a default addition to the plan library containing the strategies described in Section 6.6 as AgentSpeak(L) plans, as well as the set of meta-level actions described in Section 6.6.2. Moreover, this system was used in the implementation of the cleaning robot example described in this chapter. An important characteristic of our prototype is that it requires no knowledge from a designer that there is an underlying normative processing mechanism, that is, the designer can implement an agent using his or her regular development methodology since the mechanism we implemented imposes no restrictions on how existing plans are structured.

[1]The implementation is available for download at www.meneguzzi.eu/felipe/software.html.

Chapter 7

Conclusion

Practical agent programming languages make up a rapidly developing subfield within agent systems research, yet there are still many unresolved problems. In this thesis we have addressed several of these problems, and have advanced the state-of-the-art in clear and well-defined ways through the contributions in this thesis. Bringing the different aspects of the thesis together in this chapter, we step back and review the thesis more generally, outlining our contributions and examining limitations and possibilities for future work. First, we summarise our efforts over the various chapters in Section 7.1, reviewing the main aspects of each chapter. Then, we restate the contributions of our work in Section 7.2, taking note of the limitations we identified in our work, and pointing towards avenues for further research in Section 7.3. Finally, we make some concluding remarks in relation to the thesis in Section 7.4.

7.1 Thesis summary

Agent-based software is an established method for modelling an increasingly important number of network-centred systems. Unlike traditional object-oriented approaches to system modelling, agents are able to control their own internal state and behaviour, and have dynamic relationships among themselves and the environment in which they operate.

In order to effectively operate in an uncertain environment populated by heterogeneous agents, an agent needs not only to cooperate with unforeseen partners, but also to use its ability to control its internal operation to adapt its behaviour to the changing environmental circumstances. To facilitate the development of this kind of system, various agent programming languages have been developed. Such agent languages have often been based on a certain type of agent model that, in principle, allows for a wide range of autonomous behaviour, but these languages seldom allow the full range of behaviours envisioned in all theoretical models to be realised in practice. This is especially true for languages based on some BDI models (*e.g.* Rao's BDI model [Rao, 1996]), where aspects such as means-ends reasoning, and meta-reasoning, for example, are simply not considered. For agent programming languages to be truly useful in the development of complete agent systems (and also to be adopted more widely for real systems development), they must be able to implement agents that can adapt their behaviour and cooperation with previously unknown partners. These capabilities are often touted as being the distinguishing factor between traditional systems and agent-based systems. However, existing agent *languages* provide little, if any, support for agent cooperation and behaviour flexibility.

In seeking to address these concerns, the chapters in this thesis have addressed a series of distinct, but related, aspects in turn. Chapter 3 describes our first step towards gearing agent languages for the development of flexible systems, in which we introduced the ability to generate new plans at runtime into a BDI-based language and its associated interpreter. To do this, we took the popular AgentSpeak(L) language and developed a translation process that takes the BDI mental components and translates them into the STRIPS planning formalism, allowing an agent to tap into a planning algorithm to generate new plans to achieve its desired states of affairs. Since planning is a computationally expensive process, we also improved the long term efficiency of our planning agent by developing a plan caching method that generates a minimum context condition for newly generated plans and adds them to the plan library for future reference, eliminating the need to create plans multiple times for recurring situations. AgentSpeak(L) is a *procedural* agent language, so the introduction of this *declarative* planning process also allows AgentSpeak(L) agents to reason about declarative goals.

Building on the notion of achieving declarative goals, we introduced a motivated reasoning mechanism into AgentSpeak(L) in Chapter 4. By ascribing motivations to an AgentSpeak(L) agent, it is possible to generate goals independently from the stream of events, allowing the agent to proactively adopt goals instead of directly responding to events in the environment.

Now, the ability to plan at runtime allows an agent to develop new strategies based on information not available at design time, which is a key capability for effective operation in an environment where agents form relationships dynamically. Thus, in Chapter 5, we leveraged the planning capability introduced earlier to create a simple cooperation mechanism. Our mechanism relies on cooperating agents sharing information about actions they are willing to execute on behalf of others, so that a planning agent can include them in their newly generated plans, allowing them to solve problems they would not individually be able to solve.

Finally, we considered problems arising from the instability caused by undesirable behaviours being generated as a consequence of either individual agent interest, or the unforeseen effects of non-scripted cooperation. In order to prevent such systems from potential instability, it is now increasingly common for agent societies to be regulated by *norms* prescribing which behaviours are permitted and which are not. This has led to the development of a number of normative frameworks aimed at monitoring and enforcing norms at the environment level [Aldewereld et al., 2006; Meneguzzi et al., 2008; Oren et al., 2008b; Vázquez-Salceda et al., 2005], but which conspicuously lack methods for dynamically adapting agents to comply with them. As a result, we developed a norm processing mechanism in Chapter 6, allowing agents to change their plan libraries at runtime to comply with norms, thus ensuring our flexible, cooperating agents operate within the normative boundaries of their environments.

7.2 Contributions

As should be clear from the summary above, in this thesis we have analysed key agent concepts, including planning, motivated reasoning, cooperation and normative reasoning and operationalised them into a traditional BDI agent language. In this process, we have solved a number of problems prevalent in the area of practical agent languages suitable for application to flexible multiagent domains. Thus, our main contributions comprise the language extensions developed throughout the thesis, and span four distinct areas: plan generation, meta-reasoning, cooperation and normative reasoning. We elaborate each of these areas in the following subsections.

7.2.1 Planning capabilities

Underpinning most of our work is the requirement that agents must be able to change their plan libraries to cope with varying circumstances, social and otherwise, in an environment. In order to allow an agent to synthesise new plans at runtime and change its behaviour, we developed AgentSpeak(PL), an AgentSpeak agent interpreter capable of generating new plans at runtime using an external planning component. In achieving this, we also provided three further contributions.

First, in order to bind the planning component to AgentSpeak(PL), we have provided a **translation mechanism** from BDI mental attitudes into a classical planning formalism and from this formalism back into mental states. This allows a BDI agent to convert its mental state into planning problems that can be solved by any planning module based on the classical planning formalism. The result of this process are plans that can be integrated into the plan library of an autonomous agent, overcoming any omissions in that library by the designer.

Our translation process is applied particularly to the AgentSpeak(L) language which, as a procedural agent language, can only reason about goals *to do*. This language benefits from the planning process not only by being able to produce new plans, but also by reasoning about declarative goals (or goals *to be*), which is the type of goal processed by classical planning. Thus, second, we have extended the AgentSpeak(L) language to handle **declarative goals** in AgentSpeak(PL).

Since the process of generating new plans is also computationally expensive, our planning extension has been designed so that it is used as a last resort when all known plans have been exhausted. When plans do need to be created, the effort spent in creating them must not be wasted, leading us to develop a plan reuse strategy based on adding new plans to the plan library with an appropriate context condition. In order to achieve this, we have thirdly developed a **context generation algorithm** that analyses newly created plans and creates a *minimum* context condition that guarantees that the associated plan can be invoked only when it has a chance of executing successfully.

7.2.2 Motivated meta-reasoning

In addition to changing an agent's behaviours through planning, we have developed a **meta-reasoning** mechanism through which an agent can adapt its *behaviour selection* in the form of motivated reasoning. This mechanism was incorporated into AgentSpeak(MPL), which is an extended AgentSpeak(PL) interpreter capable of meta-reasoning through motivations. Here, motivations are used as an abstraction to define meta-reasoning strategies, such as evaluating the subjective reward of certain plans, and assessing the importance of pursuing different goals. As a result, we have developed a language for specifying motivations within a BDI agent, as well as the corresponding processing mechanism attached to the AgentSpeak(MPL) interpreter. This language allows the specification of generic motivational functions, including motivational intensity updates, goal generation and mitigation, so that a number of meta-reasoning strategies can be defined using it.

7.2.3 Multiagent domains

Agent languages in particular are notoriously lacking with regard to support for multiagent domains, inducing developers to implement cooperation methods in a more or less *ad hoc* way. In response, we have provided a method to enable the standard BDI architecture to be used in **multi-agent domains**, with cooperation and planning across multiple agents with an underlying single-agent planner. This is accomplished by leveraging the planning capabilities of AgentSpeak(PL), and by borrowing the web services notion of proxies for use in BDI agent planning. Here, local *proxy plans*, representing the plans of others, can be used locally in the construction of a new cooperative plan, encapsulating all the communication required for delegation, resulting in a simple, yet effective cooperation technique.

7.2.4 Normative processing

Having extended BDI agent languages to allow the generation of new plans at runtime, adapting an agent's behaviours, and using this plan generation mechanism for cooperation in a dynamic society, we have considered how to constrain individual agent behaviour so as not to create instability, particularly as a result of unforeseen behaviours. In order to address this, we have provided a **norm processing** mechanism into the AgentSpeak(PL) reasoning cycle by leveraging the previous contributions of planning and communication.

To accomplish this, we analysed the expected effects of a norm on an agent's reasoning cycle, resulting in the development of a generic norm processing mechanism for BDI agents. This mechanism comprises a series of algorithms that deal with the various elements of norms such as activation and expiration, resulting in changes to ensure that an agent's plan library is compliant with the norms *accepted* by an agent.

7.3 Limitations and Future Work

Our contributions span different and complex areas of agent research and, consequently, some assumptions had to be made in order to focus our work on the issues intended to be addressed. Similarly, while we have addressed the identified problems, providing strong and valuable contributions to the development of agent languages, there are still a number of avenues for future work arising from these contributions, which could enhance the work developed in this thesis in significant ways. In this section, therefore, we begin by enumerating the limitations of our work, and identifying possibilities for further research.

7.3.1 Limitations

Environmental assumptions for planning We made two important assumptions regarding the environment in which our planning agents operate in the implementation of AgentSpeak(PL) in Chapter 3. First, the rate at which the environment changes must be slower than the capacity of the underlying planner to generate an average plan, otherwise the planning agent runs the risk of generating plans that are no longer relevant at the end of planning. This assumption is necessary to ensure that existing standalone planners can be used to perform the planning, allowing our architecture to take advantage of future advances in planning. In order for this assumption to be dropped, we would need a customised and dedicated planner, like that used in the work of Despouys and Ingrand [Despouys and Ingrand, 2000]. Secondly, we assume that our agent only plans for non-numeric domains. This is an inherited limitation associated with almost all planning algorithms, without which planning must be modelled as an optimisation problem or become intractable.

Lack of methodology for motivated agent design The language developed in Chapter 4 for using motivations as an abstraction for meta-reasoning implies that it must be used by a *designer* to describe motivations in order to create a desired meta-level strategy. However, depending on the specific meta-level strategy it may not be trivial to specify the motivations that lead to the desired behaviour, and may be necessary to provide some set of guidelines to help designers specify motivations appropriately. At the same time, issues of methodology are not trivial and constitute a vast area of research which warrants a separate research effort altogether. Any sensible such effort would require consideration of existing agent design methodologies, such as TROPOS [Bresciani et al., 2004], Prometheus [Padgham and Winikoff, 2004], MaSE [García-Ojeda et al., 2008] and Gaia [Zambonelli et al., 2003], which are outside the scope of this thesis.

Lack of distribution for cooperative planning Although we have developed a multiagent cooperation mechanism in Chapter 5, we do not take advantage of the cooperating parties to distribute the planning effort. In other words, we use centralised planning to construct distributed plans. We have done so in order to leverage our previous planning mechanism, resulting in our architecture missing potential efficiency gains from distributed planning. This could be done, for example, by leveraging the work of Iwen and Mali [Iwen and Mali, 2002] on a distributed Graphplan, but again, this is beyond the scope of our work.

Constrained to BDI architectures The BDI model has been overwhelmingly popular within agent research since it provides an intuitive model for the description of human reasoning, and can be easily transposed to agent models. As a result, a large number of agent architectures have been developed drawing inspiration from the BDI model, and important agent standards, such as the FIPA agent communication language [Foundation for Intelligent Physical Agents, 2000], use BDI to underpin their semantics. For this reason, we have placed a clear emphasis on BDI agents as the base platform of our contributions. These contributions apply directly to a specific class of BDI agent architectures that include PRS [Georgeff and Lansky, 1986] and its direct descendants such as dMARS [d'Inverno et al., 2004] and AgentSpeak(L) [Rao, 1996]. However, most of our contributions can also be applied to a more general class of BDI agent architectures that are structured around a belief base, a plan library, goals and intentions, such as JACK [Howden et al., 2001], JADEX [Pokahr et al., 2005b] and BOID [Broersen et al., 2001]. Nevertheless, the techniques in this thesis are constrained to BDI architectures, and where agents are very different, such as in reactive architectures [Agre and Chapman, 1987] or Soar [Laird et al., 1987], our contributions are not directly applicable. However, the basic ideas and concepts presented in this thesis may still be relevant, with considerable adaptation, to alternative architectural approaches.

Conflict free norms assumption The norm processing mechanism described in Chapter 6 assumes that by the time norms to change behaviour start to be processed by an agent, they have already been filtered, and consistency regarding other norms adopted by the agent has been ensured. This assumption could be dropped by leveraging existing norm consistency mechanisms such as the one by Vasconcelos *et al.* [Vasconcelos et al., 2007]. Moreover, we have assumed that norms are activated and expired just once within an agent's lifetime so that the agent does not need to keep track of multiple instances of the same norm.

7.3.2 Future Work

As stated above, there are several potential areas for further work, building on the developments described in this thesis, as enumerated here.

Plan interference Our approach to the addition of new plans to the plan library of AgentSpeak(PL) considers mainly the plan selection mechanism, while ignoring the problem of adverse effects that multiple plans executing in parallel may have. Our solution to this problem, as stated in Section 3.6.4, is to prevent multiple planner generated plans from executing simultaneously. This solution hinders the potential for parallel computation in cases where it would be possible for an agent to work on multiple disjoint goals at the same time. This could be overcome through an analysis of how the consequences of plans adopted as intentions might interfere, using algorithms similar to those developed by Thangarajah *et al.* [Thangarajah et al., 2003a] in the detection of *negative* interference among plans (that is, detecting when one executing plan jeopardises the goals of another).

Motivation and planning The addition of a model of motivations to underpin the generation of goals in autonomous agents provides a rational basis not only for adopting particular goals, but also for the subsequent selection of plans to fulfill these goals, including the actions carried out in the execution of the selected plans. This information-rich connection between key parts of the reasoning process can be exploited in the refinement of the plan selection process or even the plan generation process. Thus, explicit knowledge of what caused the adoption of a certain goal allows an agent to decide the best course of action to achieve it. The approach we have taken in integrating motivated reasoning into AgentSpeak(PL) separates the model of motivation, and the ensuing motivated goals, from the planner used by the agent to achieve motivated goals. In consequence, the planner does not have access to the model of motivation, and it is possible for the planner to generate plans that, while satisfying one motivated goal (and mitigating its associated motivation), jeopardises the accomplishment of other motivated goals.

A solution to this problem would consist of making the planner aware of the motivations underlying the goals being planned for. This type of application of motivations has been proposed by Coddington [Coddington, 2007] for the MADbot System [Coddington et al., 2005]. However, it is not clear how this can be accomplished with traditional planners such as those based on the STRIPS formalism. This issue arises because STRIPS planning is inherently *discrete* in that it can only model finite goals without any quantitative information associated to either goals or the intermediate world-states considered during the generation of a plan. We believe that work towards this kind of integration between motivated reasoning and planning might benefit from some sort of constraint satisfaction-based planner [Nareyek, 2001] in order to address the need to weight competing motivations with distinct intensity variation functions.

Generalised meta-reasoning The language developed in Chapter 4 to describe meta-level reasoning is based on the abstraction of motivation, and as such ties meta-level decisions (such as when to adopt or prioritise goals) to specific events in the motivated reasoning cycle. By contrast, many approaches to meta-level reasoning introduce specific meta-level constructs, allowing a designer to specify in great detail what meta-reasoning actions certain events warrant. Our meta-reasoning mechanism could be made more general by describing meta-level actions, such as *adopt*, *drop* and *suspend* intention, and using them in explicit meta-level plans. However, the implementation of these actions within the agent interpreter is not trivial, requiring the analysis of plan flows and interdependencies, which falls outside the scope of our work. Nevertheless, this expansion would bridge the gap between the meta-level plans originally proposed in PRS [Georgeff and Lansky, 1987] and the motivation literature.

Trust and reputation for cooperation As we have seen in Chapter 5, in order to ensure the long-term viability of the plan library of an agent using our cooperation mechanism, we provide a failure handling mechanism in Section 5.5 that removes consistently failing cooperative plans from the plan library. This constitutes a fault recovery mechanism that only takes place *after* faults occur, allowing a certain degree of inefficiency in the system. A possible alternative to this mechanism could leverage work on trust and reputation (*e.g.* [Teacy et al., 2006]) in the partner selection part of our cooperation mechanism, ensuring that only trusted partners are selected, minimising the effort wasted with unreliable partners. Although it may be straightforward to *use* trust information alongside our cooperation mechanism to select partners, deriving information about trust from the outcome of cooperative plans is not as straightforward. It is overly simplistic to conclude that an agent is untrustworthy from the fact that certain cooperative plans that included this agent failed, as the failure may have taken place due to events outside this agent's control. Therefore, in order to fully take advantage of trust and reputation mechanisms, it is necessary to study how to diagnose failures and assign *blame* to failures in cooperation.

Norm consistency and norm expressions In Chapter 6 we developed a norm processing mechanism that narrows norms into two types to allow their processing in the creation and suppression of plans within a plan library. In that mechanism, we assume that norms are kept consistent somehow in order to focus on the process of plan library modification. This process, however, is critical for a normative agent to operate properly when faced with multiple conflicting norms. For example, if an agent receives a norm prohibiting the execution of an action and another obliging it to execute the same action under certain conditions, there is no way of deciding which one of these two norms must prevail.

One way of solving conflicts such as this is to adapt the work of Vasconcelos *et al.* [Vasconcelos et al., 2007], which deals with norm consistency for an entire society, to the reasoning process of an individual agent. However, this adaptation needs to take into consideration that a single agent may be under multiple *jurisdictions*, and must decide not only between conflicting norms, but also between conflicting bodies of norms from multiple environments. One solution to break this type of impasse, in turn, is to use motivational information to prioritise norms.

7.4 Conclusions

In this thesis, we have pushed the envelope of agent programming languages by developing key features of autonomous agents into a practical agent language. We have shown that although many theoretical agent models foresee many properties necessary for autonomous behaviour, such as the ability to cooperate and to change behaviour to take advantage of new circumstances, their corresponding realisations in agent languages seldom actually implement these properties.

This is particularly true when the implementation of these properties requires the integration of techniques from classic AI, such as planning and meta-reasoning. Our work has integrated such techniques into an agent language, providing a practical platform for developing autonomous agents that can change their sets of plans at runtime and discover new ways of accomplishing their objectives, both individually and cooperatively. We have also shown how desirable behaviour can be attained in a multiagent system by enabling agents to alter their plan library to comply with norms. By providing agent languages with such capabilities, we have demonstrated very concretely how agents can be used as practical receptacles of artificial reasoning. Thus, while many AI techniques are computationally expensive, we have argued in this thesis that they can be pragmatically employed in autonomous agents to solve problems not foreseen by an agent designer before system deployment, leading the way to truly adaptable systems.

Appendix A

Important planning algorithms

The actual planning algorithm encapsulated by the planning function of Definition 3.5 can vary significantly, covering any technique able to generate a plan to achieve a conjunction of literals. We have described a STRIPS-like planning formalism in Section 3.2.1, but have chosen to use modern representatives of propositional planners in our prototype, the most important of which we describe in this appendix, namely graph-based planning and planning through satisfiability testing, which we proceed to describe in Sections A.1 and A.2. Moreover, details of the conversion process from AgentSpeak to the specific planning language are also abstracted away from the agent so that different planning formalisms can be used, such as hierarchical task decomposition, described in Section A.3.

A.1 Graphplan

Graphplan is a planning algorithm based on the construction of and search in a graph [Blum and Furst, 1997]. It is considered a breakthrough in terms of efficiency regarding previous approaches to planning [Hoffmann and Nebel, 2001; Weld, 1999], and has been refined into a series of other, more powerful planners, such as IPP[1] [Köhler et al., 1997] and STAN[2] [Long and Fox, 1999], whose efficiency has been empirically verified in several planning algorithm competitions [Ghallab et al., 2002; Long and Fox, 2000].

Planning in Graphplan is based on the concept of a *planning graph*, which is a data structure in which information regarding the planning problem is stored in a directed and levelled graph in such a way that the search for a solution can be optimised. The planning graph is not a state-space graph in which a plan is a path through the graph. Instead, a plan in the planning graph is essentially a flow in the network flow sense, which is composed of more

[1]Interference Progression Planner
[2]State Analysis

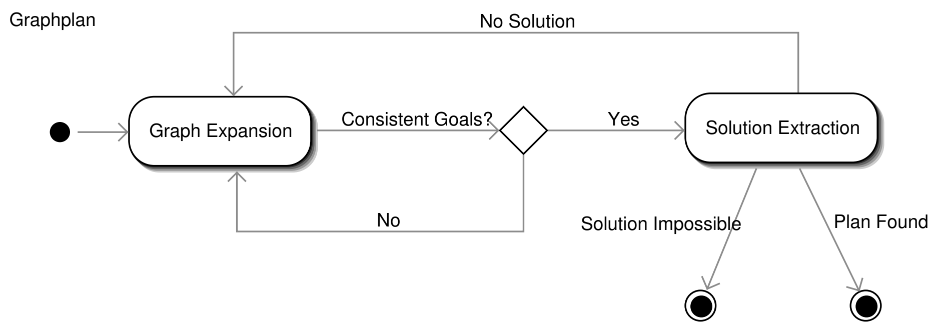


FIGURE A.1: Overview of the Graphplan algorithm.

than one path in a directed graph with the added constraint that paths must not include mutually exclusive nodes in the same graph level.

The planning graph consists of alternating levels of instantiated operators and propositions representing temporally ordered actions and the world-states that occur between the execution of these actions. Proposition levels contain nodes labelled with literals, and are connected to the actions in the subsequent action level through precondition arcs. Action nodes are labelled with operators and are connected to the nodes in the subsequent proposition nodes by effect arcs. Every proposition level denotes literals that are possibly true at a given time step, so the first proposition level represents the literals that are possibly true before plan execution (*e.g.* time step 1), the next proposition level represents the literals that are possibly true at the next time step (*e.g.* time step 2) and so forth. Action levels denote operators that can be executed at a given moment in time in such a way that the first action level represents the operators that may be executed at time step 1 and so forth. The graph contains mutual exclusion relations (*mutex*) between nodes in the same graph level, denoting that two nodes connected by a *mutex* arc cannot be simultaneously present in the same graph level for any solution. These mutual exclusion relations play a key role in the efficiency of the algorithm, as the search for a solution can completely ignore any flows that include mutually exclusive nodes in a given level.

Construction of this graph is efficient, having polynomial complexity for both graph size and construction time regarding problem size [Blum and Furst, 1997]. This graph is then used by the planner in the search for a solution to the planning problem using data regarding the relations between operators and states to speed up the search. The basic Graphplan algorithm (*i.e.* without the optimisations proposed by other researchers) is divided into two distinct phases: graph expansion and solution extraction. The algorithm alternates execution of graph expansion and solution extraction until a solution is found or it is proven that no solution exists, as illustrated in Figure A.1.

A.2 SAT Planning

Planning through *satisfiability testing* (SAT) follows the experiments made by Kautz and Selman [Kautz and Selman, 1992] on the compilation of STRIPS-based planning problems into logical formulas to be solved by programs that test the satisfiability of the generated formulas. These experiments were motivated by performance improvements in the area of propositional satisfiability verification [Cook and Mitchel, 1997].

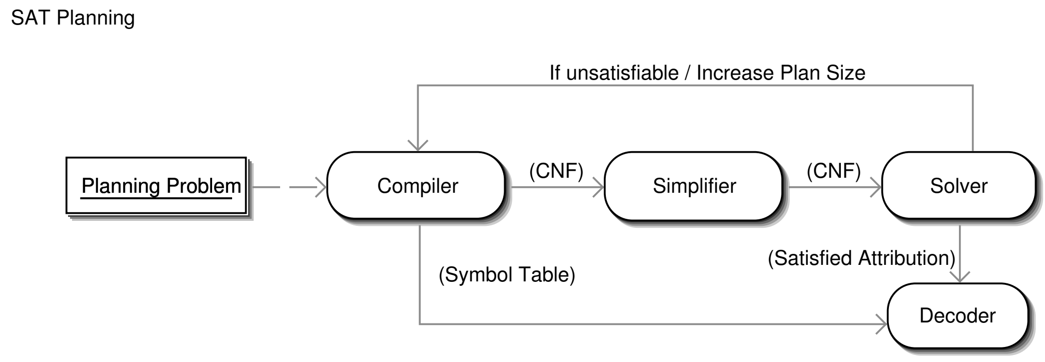


Figure A.2: Activities of a typical SAT planner.

A typical SAT planner starts by taking the input planning problem, estimating a plan size, and generating formula in propositional logic that, if satisfiable, implies the existence of a plan that represents the solution to the input problem [Kautz and Selman, 1996]. A *symbol table* stores the correspondence between the propositional variables created during compilation and the planning problem received as input. A quick simplification process is used to reduce the formula created by the compiler into *conjunctive normal form* (CNF). The solver generally consists of an off-the-shelf satisfiability tester that tries to find a correct assignment for the variables in the compiled formula. If such an assignment is found, the decoder translates this variable assignment, using the symbol table, into a plan. If the solver determines that the formula is unsatisfiable, then the compiler generates a new encoding reflecting a longer plan. This process of systematic generation of increasingly longer propositional encodings is repeated until a satisfiable assignment is found, or a stop condition is reached (resulting in the failure to find a solution). The typical architecture of a SAT planner is illustrated in Figure A.2.

A.3 Hierarchical Task Network Decomposition

Hierarchical task network (HTN) decomposition has evolved as an alternative approach to planning [Erol et al., 1994], departing from the STRIPS specification in that a planning problem representation is enriched with a hierarchy of goals and methods for refining these goals into concrete actions.

Task networks correspond to a set of *tasks* that need to be accomplished, as well as constraints on how these tasks must be carried out. Tasks are represented similarly to operator headers (*e.g.* move(A,B)), with parameters that may consist of variables and constants. Tasks may be *non-primitive*, requiring the planner to refine them further into *primitive* tasks, which can be directly executed and are analogous constructs to STRIPS-like operators. An HTN planning problem must also include a set of *methods* specifying how to perform non-primitive tasks, along with constraints for the application of these methods. These methods allow the planner to refine a *task* into a more detailed *task network* if this task can be replaced by the *task network* while respecting some set of constraints.

Planning in an HTN planner starts with a task network describing *goal tasks*, which are analogous to STRIPS goals, and attempting to substitute tasks in the network until only primitive tasks remain, allowing the execution of concrete operators. This process is similar to how grammar rules are applied to generate strings.

Bibliography

Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, 1987.

Rachid Alami, Sara Fleury, Matthieu Herrb, Francóis Félix Ingrand, and Frédéric Robert. Multi-robot cooperation in the MARTHA project. *IEEE Robotics and Automation Magazine: Robotics & Automation in the European Union*, 5(1):36–47, 1998.

Huib Aldewereld, Frank Dignum, Andrés García-Camino, Pablo Noriega, Juan Antonio Rodríguez-Aguilar, and Carles Sierra. Operationalisation of norms for usage in electronic institutions. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 223–225, New York, NY, USA, 2006. ACM.

Jose A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 83–88, St Paul, MN, USA, 1988. American Association for Artificial Intelligence.

Davide Ancona, Viviana Mascardi, Jomi F. Hübner, and Rafael H. Bordini. Coo-agentspeak: Cooperation in agentspeak through plan exchange. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 696–705, 2004.

John L. Austin. *How To Do Things With Words*. Oxford University Press, Oxford, 1962.

Christian Balkenius. The roots of motivation. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 513–523, 1993.

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

Rafael H. Bordini, Ana L. C. Bazzan, Rafael de O. Jannone, Daniel M. Basso, Rosa M. Vicari, and Victor R. Lesser. AgentSpeak(XL): efficient intention selection in BDI agents via decision-theoretic task scheduling. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1294–1302, 2002.

Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer-Verlag, 2005a.

Rafael H. Bordini, Jomi Fred Hübner, and Renata Vieira. Jason and the golden fleece of agent-oriented programming. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 3–37. Springer-Verlag, 2005b.

Rafael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason.* Wiley, 2007.

Michael E. Bratman. Two faces of intention. *Philosophical Review*, 93:375–405, 1984.

Michael E. Bratman. *Intention, Plans and Practical Reason.* Harvard University Press, Cambridge, MA, 1987.

Lars Braubach, Alexander Pokahr, Winifried Lamersdorf, and Daniel Moldt. Goal representation for BDI agent systems. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Proceedings of the 2nd International Workshop on Programming Multiagent Systems Languages and Tools*, pages 7–9, 2004.

Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, May 2004.

Jan Broersen, Mehdi Dastani, Joris Hulstijn, Zisheng Huang, and Leendert van der Torre. The boid architecture: conflicts between beliefs, obligations, intentions and desires. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 9–16, 2001.

Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

Dolores Cañamero. Modeling motivations and emotions as a basis for intelligent behavior. In *Proceedings of the First International Conference on Autonomous Agents*, pages 148–155, 1997.

Lucas Carlson and Leonard Richardson. *Ruby Cookbook: recipes for object-oriented scripting.* O'Reilly, 2006.

Alexandra Coddington. Integrating motivations with planning. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1–3, New York, NY, USA, 2007. ACM.

Alexandra M. Coddington, Maria Fox, Jonathan Gough, Derek Long, and Ivan Serina. MADbot: a motivated and goal directed robot. In *Proceedings of the 20th National Conference on Artificial Intelligence*, pages 1680–1681, Menlo Park, California, 2005. AAAI Press.

Phillip R. Cohen and Hector J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(2-3):213–261, 1990.

Stephen A. Cook and David G. Mitchel. Finding hard instances of the satisfiability problem: A survey. In Dingzhu Du, Jun Gu, and Panos M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 11–13. American Mathematical Society, Providence, RI, 1997.

Jeffrey S. Cox, Edmund H. Durfee, and Thomas Bartold. A distributed framework for solving the multiagent plan coordination problem. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 821–827. ACM Press, 2005.

Michael Cox and Anita Raja. Metareasoning: A manifesto. In *Proceedings of AAAI 2008 Workshop on Metareasoning: Thinking about Thinking*, 2008.

Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Ch. Meyer. A programming language for cognitive agents goal directed 3APL. In *Proceedings of the International Workshop on Programming Multiagent Systems Languages and Tools*, volume 3067 of *LNCS*, pages 111–130. Springer-Verlag, 2004.

Mehdi Dastani, M. Birna van Riemsdijk, and John-Jules Ch. Meyer. *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 2: Programming Multi-Agent Systems in 3APL, pages 39–68. Springer-Verlag, 2005.

Etienne de Sevin and Daniel Thalmann. An affective model of action selection for virtual humans. In *Proceedings of Agents that Want and Like: Motivational and Emotional Roots of Cognition and Action; Artificial Intelligence and Social Behaviors 2005 Conference*, pages 110–113, 2005a.

Etienne de Sevin and Daniel Thalmann. A motivational model of action selection for virtual humans. In *Computer Graphics International 2005*, pages 213–220, 2005b.

Keith S. Decker. Task environment centered simulation. In *Simulating organizations: computational models of institutions and groups*, pages 105–128, Cambridge, MA, USA, 1998. MIT Press.

Marie E. desJardins, Edmund H. Durfee, Charles L. Ortiz Jr., and Michael J. Wolverton. A survey of research in distributed, continual planning. *AI Magazine*, 20(4):13–22, 1999.

Olivier Despouys and François Felix Ingrand. Propice-plan: Toward a unified framework for planning and execution. In *Proceedings of the 5th European Conference on Planning*, pages 278–293, London, UK, 2000. Springer-Verlag.

Frank Dignum. Autonomous agents with norms. *Artificial Intelligence and Law*, 7(1):69–79, March 1999.

Mark d'Inverno and Michael Luck. Engineering AgentSpeak(L): A formal computational model. *Journal of Logic and Computation*, 8(3):233–260, 1998.

Mark d'Inverno, Michael Luck, Michael Georgeff, David Kinny, and Michael Wooldridge. The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System. *Autonomous Agents and Multi-Agent Systems*, 9(1 - 2):5–53, July 2004.

Mark d'Inverno, Michael Luck, and Michael Wooldridge. Cooperation structures. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 600–605, 1997.

J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman. On cooperation in multi-agent systems. *Knowledge Engineering Review*, 12(3):309–314, 1997.

Simon Duff, James Harland, and John Thangarajah. On proactivity and maintenance goals. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1033–1040, 2006.

Edmund H. Durfee. *Coordination of Distributed Problem Solvers.* Kluwer Academic Publishers, 1988.

Edmund H. Durfee. Distributed problem solving and planning. In *Multi-agents systems and applications*, pages 118–149, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

Edmund H. Durfee and V. R. Lesser. Predictability versus responsiveness: coordinating problem solvers in dynamic domains. In *Readings in uncertain reasoning*, pages 198–203, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

E.H. Durfee and V.R. Lesser. Partial global planning: a coordination framework for distributed hypothesis formation. *IEEE Transactions on Systems, Man and Cybernetics*, 21 (5):1167–1183, 1991.

Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1123–1128, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

Marc Esteva, Julian A. Padget, and Carles Sierra. Formalizing a language for institutions and norms. In John-Jules Ch. Meyer and Milind Tambe, editors, *Intelligent Agents VIII, 8th International Workshop*, volume 2333 of *LNCS*, pages 348–366. Springer-Verlag, 2001.

Noura Faci, Sanjay Modgil, Nir Oren, Felipe Meneguzzi, Simon Miles, and Michael Luck. Towards a monitoring framework for agent-based contract systems. In Matthias Klusch, Michal Pechoucek, and Axel Polleres, editors, *Proceedings of the Twelfth International Workshop on Cooperative Information Agents*, 2008.

Innes A. Ferguson. Integrated control and coordinated behaviour: a case for agent models. In Michael J. Wooldridge and Nicholas R. Jennings, editors, *Intelligent Agents*, volume 890 of *LNCS*, pages 203–218. Springer-Verlag, 1995.

Richard Fikes and Nils Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management*, pages 456–463. ACM, 1994.

Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. http://www.fipa.org, 2000. SC00061.

Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Intelligent Agents III*, volume 1193 of *LNCS*, Berlin, Germany, 1996. Springer-Verlag.

A. Garcia-Camino, P. Noriega, and J. A. Rodriguez-Aguilar. Implementing norms in electronic institutions. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 667–673. ACM Press, 2005.

Juan C. García-Ojeda, Scott A. DeLoach, Robby, Walamitien H. Oyenan, and Jorge Valenzuela. O-mase: A customizable approach to developing multiagent development processes. In Michael Luck and Lin Padgham, editors, *Agent-Oriented Software Engineering VIII*, volume 4951 of *LNCS*, pages 1–15. Springer-Verlag, 2008.

Michael R. Genesereth and Richard E. Fikes. Knowledge interchange format, version 3.0 reference manual. Technical Report Logic-92-1, Stanford University, Stanford, California, 1992.

Michael P. Georgeff and François Félix Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 972–978, Detroit, MI, USA, 1989a. Morgan Kaufmann.

Michael P. Georgeff and François Félix Ingrand. Monitoring and control of spacecraft systems using procedural reasoning. In *Proceedings of the Space Operations and Robotics Workshop*, Houston, USA, 1989b.

Michael P. Georgeff and Amy L. Lansky. Procedural knowledge. *Proceedings of the IEEE, Special Issue on Knowledge Representation*, 74(10):1383–1898, 1986.

Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of the American Association for Artificial Intelligence (AAAI)*, pages 677–682, Seattle, WA, USA, 1987. Morgan Kaufmann Publishers.

Malik Ghallab, Joachim Hertzberg, and Paolo Traverso. *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems.* AAAI, Toulouse, France, April 23-27 2002.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice.* Elsevier, 2004.

Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

Stephen Grand and Dave Cliff. Creatures: Entertainment software agents with artificial life. *Autonomous Agents and Multi-Agent Systems*, 1(1):39–57, 1998.

Nathan Griffiths and Michael Luck. Cooperative plan selection through trust. In *Proceedings of the Ninth European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1647 of *LNAI*, pages 162–174, 1999.

Nathan Griffiths and Michael Luck. Coalition formation through motivation and trust. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 17–24. ACM Press, 2003.

William Grosso. *Java RMI: Designing & Building Distributed Applications.* O'Reilly, 2002.

Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3): 251–321, 1985.

Koen V. Hindriks, Frank S. De Boer, Wiebe Van der Hoek, and John-Jules Ch. Meyer. Agent programming in 3APL. *International Journal of Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.

Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming with declarative goals. In *Intelligent Agents VII. Agent Theories Architectures and Languages, Seventh International Workshop*, volume 1986 of *LNCS*, pages 228–243. Springer-Verlag, 2001.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Nick Howden, Ralph Rönnquist, Andrew Hodgson, and Andrew Lucas. Jack: Summary of an agent infrastructure. In *Proceedings of the 5th International Conference on Autonomous Agents*, Montreal, Canada, 2001.

Jomi Fred Hübner, Rafael H. Bordini, and Michael Wooldridge. Plan patterns for declarative goals in agentspeak. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1291–1293, 2006.

Jomi Fred Hübner, Rafael H. Bordini, and Michael Wooldridge. Programming declarative goals using plan patterns. In Matteo Baldoni and Ulle Endriss, editors, *Proceedings of the Fourth Workshop on Declarative Agent Languages and Technologies*, volume 4327 of *LNCS*, pages 123–140. Springer-Verlag, 2006.

Félix Ingrand and Olivier Despouys. Extending procedural reasoning toward robot actions planning. In *Proceedings of the 2001 IEEE International Conference on Robotics and Automation*, pages 9–10, Seoul, Korea, 2001.

François F. Ingrand, Michael P. Georgeff, and Anand S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert, Knowledge-Based Diagnosis in Process Engineering*, 7(6):33–44, 1992.

Mark Iwen and Amol Dattatraya Mali. Distributed graphplan. In *Proceedingds of the 14th IEEE International Conference on Tools with Artificial Intelligence*, pages 138–145, Washington, DC, USA, 2002. IEEE Computer Society.

Nicholas R. Jennings. On agent-based software engineering. *Artificial Intelligence*, 117(2): 277–296, 2000.

Nicholas R. Jennings, Anthony G. Cohn, Maria Fox, Derek Long, Michael Luck, Danius T. Michaelides, Steve Munroe, and Mark J. Weal. *Cognitive Systems: Information Processing Meets Brain Science*, chapter 8. Motivation, Planning and Interaction, pages 163–188. Queen's Printer and Controller of HMSO, 2006.

Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

Andrew J. I. Jones and Ingmar Pörn. 'ought' and 'must'. *Synthese*, 66(1):89–93, 1986.

Dionysis Kalofonos and Timothy J. Norman. An investigation into team-based planning. In *2004 IEEE International Conference on Systems, Man and Cybernetics*, pages 5590–5595, 2004.

Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence*, pages 359–363, Chichester, UK, 1992. Wiley.

Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, August 1996. AAAI Press / MIT Press.

Jana Köhler. Solving complex planning tasks through extraction of subproblems. In Reid Simmons, Manuela Veloso, and Stephen Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 62–69, Pittsburgh, PA, USA, 1998. AAAI Press.

Jana Köhler, Bernhard Nebel, Joerg Hoffmann, and Yannis Dimopoulos. Extending planning graphs to an ADL subset. In S. Steel, editor, *Proceedings of the 4th European Conference on Planning*, volume 1348 of *Lecture Notes in Computer Science*, pages 273–285. Springer-Verlag, Germany, 1997.

Martin J. Kollingbaum and Timothy J. Norman. Norm adoption and consistency in the noa agent architecture. In *Programming Multi-Agent Systems*, volume 3067 of *LNCS*, pages 169–186, 2003.

John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

V. Lesser, K. Decker, N. Carver, D. Neiman, M. N Prasad, and T. Wagner. Evolution of the gpgp domain-independent coordination framework. Technical report, Amherst, MA, USA, 1998.

V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. NagendraPrasad, A. Raja, R. Vincent, P. Xuan, and X.Q Zhang. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. *Autonomous Agents and Multi-Agent Systems*, 9(1):87–143, July 2004.

Alessio Lomuscio and Marek Sergot. Deontic interpreted systems. *Studia Logica*, 75(1): 63–92, 2003.

Derek Long and Maria Fox. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research (JAIR)*, 10:87–115, 1999.

Derek Long and Maria Fox. Automatic synthesis and use of generic types in planning. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, pages 196–205, Breckenridge, CO, USA, 2000. AAAI Press.

Fabiola Lopez y Lopez. *Social Power and Norms: Impact on agent behaviour.* PhD thesis, University of Southampton, 2003.

Fabiola Lopez y Lopez and Michael Luck. Modelling norms for autonomous agents. In *Proceedings of the Fourth Mexican International Conference on Computer Science*, pages 238–245, 2003.

Fabiola Lopez y Lopez, Michael Luck, and Mark d'Inverno. Normative agent reasoning in dynamic societies. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 732–739. IEEE Computer Society, 2004.

Fabiola Lopez y Lopez, Michael Luck, and Mark d'Inverno. A normative framework for agent-based systems. In *Proceedings of the First International Symposium on Normative Multi-Agent Systems*, 2005.

Michael Luck, Ronald Ashri, and Mark d'Inverno. *Agent-Based Software Development.* Artech House, 2004.

Michael Luck and Mark d'Inverno. Motivated behavior for goal adoption. In Springer-Verlag, editor, *Selected Papers from the 4th Australian Workshop on Distributed Artificial Intelligence, Multi-Agent Systems*, pages 58–73, 1998.

Michael Luck, Steve Munroe, and Mark d'Inverno. *Autonomy: Variable and Generative*, chapter Chapter 2, pages 9–22. Kluwer, 2003.

Sheila A. McIlraith and Ronald Fadel. Planning with complex actions. In Salem Benferhat and Enrico Giunchiglia, editors, *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning*, pages 356–364, 2002.

Alfred R. Mele. *Motivation and Agency.* Oxford University Press, 2003.

Felipe Meneguzzi and Michael Luck. Composing high-level plans for declarative agent programming. In *Proceedings of the Fifth Workshop on Declarative Agent Languages*, pages 115–130, 2007a.

Felipe Meneguzzi and Michael Luck. Motivations as an abstraction of meta-level reasoning. In Hans-Dieter Burkhard, Gabriela Lindemann, Rineke Verbrugge, and László Z. Varga, editors, *Proceedings of the 5th International Central and Eastern European Conference on Multi-Agent Systems*, volume 4696 of *LNAI*, pages 204–214. Springer-Verlag, 2007b.

Felipe Meneguzzi and Michael Luck. Interaction among agents that plan. In Bernhard Jung, Fabien Michel, Alessandro Ricci, and Paolo Petta, editors, *Proceedings of the Sixth International Workshop: From Agent Theory to Agent Implementation*, pages 133–140, 2008a.

Felipe Meneguzzi and Michael Luck. Leveraging new plans in AgentSpeak(PL). In Matteo Baldoni, Tran Cao Son, M. Birna van Riemsdijk, and Michael Winikoff, editors, *Proceedings of the Sixth Workshop on Declarative Agent Languages*, pages 63–78, 2008b.

Felipe Meneguzzi and Michael Luck. Norm-based behaviour modification in BDI agents. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 177–184, 2009.

Felipe Meneguzzi, Simon Miles, Camden Holt, Michael Luck, Nir Oren, Sanjay Modgil, Nora Faci, and Martin Kollingbaum. Electronic contracting in aircraft aftercare: A case study. In *Proceedings of the 7th International Conference on Autonomous Agents and Multiagent Systems*, pages 63–70, 2008.

Felipe Meneguzzi, Avelino Francisco Zorzo, Michael da Costa Móra, and Michael Luck. Incorporating planning into BDI agents. *Scalable Computing: Practice and Experience*, 8:15–28, 2007.

Felipe Rech Meneguzzi, Avelino Francisco Zorzo, and Michael Da Costa Móra. Propositional planning in BDI agents. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 58–63, Nicosia, Cyprus, 2004. ACM Press.

Nicolas Meuleau and David E. Smith. Optimal limited contingency planning. In *Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence*, pages 417–426, Acapulco, Mexico, 2003.

Marvin Minsky. *The society of mind.* Simon & Schuster, Inc., New York, NY, USA, 1986.

Andrew H. Mishkin, Jack C. Morrison, Tam T. Nguyen, Henry W. Stone, Brian K. Cooper, and Brian H. Wilcox. Experiences with operations and autonomy of the mars pathfinder microrover. *Aerospace Conference, 1998. Proceedings., IEEE*, 2:337–351, 1998.

David Moffat and Nico H. Frijda. Where there's a will there's an agent. In *Proceedings of the Workshop on Agent Theories, Architectures, and Languages on Intelligent agents*, volume 890 of *LNCS*, pages 245–260, New York, USA, 1995. Springer-Verlag.

Michael da Costa Móra, José Gabriel Pereira Lopes, Rosa Maria Vicari, and Helder Coelho. BDI models and systems: Bridging the gap. In *Intelligent Agents V, Agent Theories, Architectures, and Languages, Fifth International Workshop*, volume 1555 of *LNCS*, pages 11–27. Springer-Verlag, 1999.

Álvaro F. Moreira, Renata Vieira, and Rafael H. Bordini. Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In João Alexandre Leite, Andrea Omicini, Leon Sterling, and Paolo Torroni, editors, *Proceedings of the First Workshop on Declarative Agent Languages and Technologies*, volume 2990 of *LNCS*, pages 135–154. Springer-Verlag, 2003.

Philippe Morignot and Barbara Hayes-Roth. Motivated agents. Technical report, Knowledge Systems Laboratory – Stanford University, 1996.

Jörg Müller, Markus Pischel, and Michael Thiel. Modelling reactive behaviour in vertically layered agent architectures. In Michael J. Wooldridge and Nicholas R. Jennings, editors, *Intelligent Agents*, volume 890 of *LNCS*, pages 267–276. Springer-Verlag, 1995.

Steve Munroe, Michael Luck, and Mark d'Inverno. Motivation-based selection of negotiation partners. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1520–1521, 2004.

Alexander Nareyek. Beyond the plan-length criterion. In *Proceedings of the Workshop on Local Search for Planning and Scheduling-Revised Papers*, volume 2148 of *LNAI*, pages 55–78. Springer-Verlag, 2001.

Bernhard Nebel. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, 12:271–315, 2000.

Ulf Nilsson and Jan Maluszynski. *Logic, Programming and Prolog.* John Wiley & Sons Ltd., 1995.

Timothy J. Norman and Derek Long. Goal creation in motivated agents. In *Intelligent Agents*, volume 890 of *LNCS*, pages 277–290. Springer-Verlag, 1994.

Timothy J. Norman and Derek Long. Alarms: An implementation of motivated agency. In *Intelligent Agents II*, volume 1037 of *LNCS*, pages 219–234. Springer-Verlag, 1995.

Timothy J. Norman, Alun Preece, Stuart Chalmers, Nicholas R. Jennings, Michael Luck, Viet D. Dang, Thuc D. Nguyen, Vikas Deora, Jianhua Shao, W. Alex Gray, and Nick J. Fiddian. Agent-based formation of virtual organisations. *Knowledge-Based Systems*, 17 (2-4):103–111, May 2004.

Timothy J. Norman and Chris Reed. Delegation and responsibility. In Cristiano Castelfranchi and Yves Lespérance, editors, *Intelligent Agents VII*, volume 1986 of *Lecture Notes in Computer Science*, pages 136–149. Springer-Verlag, 2001.

Hyacinth S. Nwana. Software agents: An overview of software agents. *Knowledge Engineering Review*, 1996.

James Odell. Objects and agents compared. *Journal of Object Technology*, 1(1):41–53, 2002.

Nir Oren, Michael Luck, and Timothy J. Norman. Argumentation for normative reasoning. In *Proceedings of the Symposium on Behaviour Regulation in Multi-Agent Systems*, pages 55–60, 2008a.

Nir Oren, Sofia Panagiotidi, Javier Vazquez-Salceda, Sanjay Modgil, Michael Luck, and Simon Miles. Towards a formalisation of electronic contracting environments. In *Proceedings of Coordination, Organization, Institutions and Norms in Agent Systems, the International Workshop at AAAI 2008*, pages 61–68, Chicago, Illinois, USA, 2008b.

Lin Padgham and Michael Winikoff. *Developing Intelligent Agent Systems: A Practical Guide.* Wiley, 2004.

Edwin P. D. Pednault. Adl: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

Marco Pistore, Annapaola Marconi, Piergiorgio Bertoli, and Paolo Traverso. Automated composition of web services by planning at the knowledge level. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 1252–1259, 2005.

Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. A goal deliberation strategy for BDI agent systems. In *Proceedings of the Third German Conference on Multiagent System Technologies*, pages 82–93, 2005a.

Alexander Pokahr, Lars Braubach, and Winfried Lamersdorf. Jadex: A BDI reasoning engine. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah-Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 149–174. Springer-Verlag, 2005b.

Ingmar Pörn. *Logic of Power.* Blackwell Publishers, 1970.

Anita Raja and Victor Lesser. Meta-level reasoning in deliberative agents. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 141–147, 2004.

Anand S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In Walter Van de Velde and John W. Perram, editors, *Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, volume 1038 of *LNCS*, pages 42–55. Springer-Verlag, 1996.

Anand S. Rao and Michael P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, pages 312–319, San Francisco, 1995a.

Anand S. Rao and Michael P. Georgeff. Formal models and decision procedures for multi-agent systems. Technical Report 61, Australian Artificial Intelligence Institute, 1995b. Technical Note.

Sebastian Sardiña, Lavindra de Silva, and Lin Padgham. Hierarchical Planning in BDI Agent Programming Languages: A Formal Approach. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1001–1008, 2006.

Sebastian Sardiña and Lin Padgham. Goals in the context of BDI plan failure and planning. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 16–23, 2007.

Martijn Schut and Michael Wooldridge. The control of reasoning in resource-bounded agents. *The Knowledge Engineering Review*, 16(3):215–240, 2001.

John R. Searle. *Speech Acts : An Essay in the Philosophy of Language.* Cambridge University Press, 1969.

Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

Yoav Shoham and Moshe Tennenholtz. On Social Laws for Artificial Agent Societies: Off-Line Design. *Artificial Intelligence*, 73(1-2):231–252, 1995.

Munindar P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.

Kevin Smith. Clerks. Miramax Films, October 1995.

Catherine Soanes and Sara Hawker, editors. *Compact Oxford English Dictionary of Current English*. Oxford University Press, 3rd edition, 2005.

Luc Steels. A case study in the behavior-oriented design of autonomous agents. In Dave Cliff, Philip Husbands, Jean-Arcady Meyer, and Stewart W. Wilson, editors, *From Animals to Animats*, pages 445–452, Cambridge, MA, USA, 1994.

William Thomas Luke Teacy, Jigar Patel, Nicholas R. Jennings, and Michael Luck. Travos: Trust and reputation in the context of inaccurate information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 12(2):183–198, 2006.

John Thangarajah, James Harland, David Morley, and Neil Yorke-Smith. Aborting tasks in BDI agents. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 8–15, 2007.

John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting & avoiding interference between goals in intelligent agents. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 721–726, 2003a.

John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting & exploiting positive goal interaction in intelligent agents. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 401–408, 2003b.

S Rebecca Thomas. The placa agent programming language. In Michael J. Wooldridge and Nicholas R. Jennings, editors, *Intelligent Agents*, volume 890 of *LNCS*, pages 355–370. Springer-Verlag, 1995.

Alan Matheson Turing. Computing machinery and intelligence. *Mind*, 59:433–460, 1950.

Birna van Riemsdijk, Wiebe van der Hoek, and John-Jules Ch. Meyer. Agent programming in dribble: from beliefs to goals using plans. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 393–400, Melbourne, Australia, 2003. ACM Press.

M. Birna van Riemsdijk, Mehdi Dastani, and John-Jules Ch. Meyer. Semantics of declarative goals in agent programming. In *Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 133–140, 2005.

Wamberto Vasconcelos, Martin J. Kollingbaum, and Timothy J. Norman. Resolving conflict and inconsistency in norm-regulated virtual organizations. In *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1–8. ACM, 2007.

Javier Vázquez-Salceda, Huib Aldewereld, and Frank Dignum. Norms in multiagent systems: from theory to practice. *International Journal of Computer Systems Science & Engineering*, 20(4):225–236, 2005.

Renata Vieira, Álvaro F. Moreira, Michael Wooldridge, and Rafael H. Bordini. On the formal semantics of speech-act based communication in an agent-oriented programming language. *Journal of Artificial Intelligence Research*, 29:221–267, 2007.

T. Wagner, A. Garvey, and V. Lesser. Complex goal criteria and its application in design-to-criteria scheduling. Technical report, Amherst, MA, USA, 1997.

Thomas Wagner and Victor R. Lesser. Relating quantified motivations for organizationally situated agents. In *Intelligent Agents VI*, volume 1757 of *LNAI*, pages 334–348, London, UK, 2000. Springer-Verlag.

Andrzej Walczak, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. Augmenting BDI Agents with Deliberative Planning Techniques. In *Proceedings of the Fifth International Workshop on Programming Multiagent Systems*, 2006.

Peter Wegner and Dina Goldin. Computation beyond turing machines. *Communications of the ACM*, 46(4):100–102, 2003.

Eric W. Weisstein. Mathworld: Power set. Web, 1999. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PowerSet.html.

Daniel S. Weld. Recent Advances in AI Planning. *AI Magazine*, 20(2):93–123, 1999.

Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. Declarative & Procedural Goals in Intelligent Agent Systems. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *Proceedings of the Eighth International Conference on Principles and Knowledge Representation and Reasoning*, pages 470–481. Morgan Kaufmann, 2002.

Michael Wooldridge. *Intelligent Agents*, chapter 2. The MIT Press, 1999.

Michael Wooldridge. *An Introduction to Multiagent Systems.* John Wiley & Sons, 2002.

Michael Wooldridge and Nicholas Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge. Developing multiagent systems: the gaia methodology. *ACM Trans on Software Engineering and Methodology*, 12(3):317–370, 2003.

Jian Feng Zhang, Xuan Thang Nguyen, and Ryszard Kowalczyk. Graph-based multi-agent replanning algorithm. In *Proceedings of the Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 793–800, 2007.