Agent Programming Languages: Programming with Mental Models

Agent Programmeertalen:
Programmeren met Mentale Modellen
(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van
doctor aan de Universiteit Utrecht
op gezag van de Rector Magnificus, Prof. Dr. H.O. Voorma,
ingevolge het besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 5 februari 2001 des middags te 4.15 uur

door

Koenraad Victor Hindriks

geboren op 29 december 1971,
te Groningen

promotor: Prof. Dr. J.-J. Ch. Meyer

co-promotoren: Dr. F.S. de Boer
Dr. W. van der Hoek

faculteit Wiskunde en Informatica, Universiteit Utrecht

Cover Design by Pieter Wesseling

ISBN 90-393-25901

SIKS Dissertation Series No. 2001-2

The research in this thesis has been carried out under the auspices of SIKS, the Dutch Graduate School for Information and Knowledge Systems.

CONTENTS

Preface

In my final year as a student at the University of Groningen, I was lucky enough to discover the research area of *intelligent agents*. At that time, I was still mainly studying theoretical subjects, like the lambda calculus and pure logic. Although I have learned to appreciate these fields very much, at that time I sometimes felt that I was lost in a mace of great complexity, and I did not know which direction to take. Intelligent agents offered a way out of the mace and a subject to write about for my masters thesis. From that moment on, I knew I wanted to get a research position as a PhD student...

During the writing of my masters thesis, I found out that at the University Utrecht Bernd van Linder, a PhD student, was also working on intelligent agents. On several occasions we exchanged ideas and I became very interested in the group of John-Jules Meyer, who supervised Bernd. I was fortunate enough to get the opportunity to work in Utrecht very soon after finishing my thesis.

The past four years I have very much enjoyed working with John-Jules, Frank de Boer, and Wiebe van der Hoek. John-Jules always has been very inspiring. His enthusiasm for research is never lacking and his vision of the future is still much bigger than mine. Frank's knowledge of concurrency theory has been invaluable for my research, and because of his typical Dutch humour we could never take our work too seriously. Wiebe made sure my work lived up to the standards of academic research, but I probably still do not pay enough attention to detail... He always was very supportive, and fun to work with.

My fellow PhD students with whom I shared a room have made live at the university much more pleasant. Rogier, who started just before me, always helped me out with any problem whatsoever. Paul, who joined us some years later, liked to enter a discussion just as much as myself. We share many interests, and our main difference has been that while Paul was writing papers about games, I was playing them on the Internet. During the last months of my research, I have enjoyed working with Wieke. She is always able to put things into a different perspective.

There are many more colleagues and other people to thank. In particular, I would like to thank Marco for all those pleasant coffee breaks. Richard always took time to discuss career opportunities, and all other most important things in live. During my PhD I have been able to meet many other people in computing science. I was fortunate enough to visit the University of Toronto for a few months and very much appreciated working together with Hector Levesque and Yves Lespérance. Of course, it also was always great to see my British colleagues again at the many agent conferences.

I want to thank the members of my reading committee: Prof. dr. Hector Levesque, Prof. dr. John-Jules Meyer, Prof. dr. Gerard Renardel de Lavelette, and Prof. dr. Jan Treur.

In science, simplicity and beauty are two of the highest goals to achieve. If I did not succeed in achieving both of these goals completely in this thesis, I am very grateful to Pieter that he did achieve them in the design of the cover of my thesis.

Finally, I would like to thank Mirjam for being there for me the past four years. She always knew what I was talking about and with her I could always share my thoughts and feelings.

Koen Hindriks
Utrecht, December 2000

CHAPTER 1

Introduction

This thesis is about agent-oriented programming. The key concept is that of an agent. Here, an agent is a computational entity that is viewed as having beliefs, goals, and plans to achieve those goals. An agent is defined as a computational entity here to stress the fact that we are interested in building or programming agent systems. We do not study human or non-software agents in this thesis (for another view on agents, see Franklin & Graesser (1997)). Instead, our aim is to introduce and discuss a new programming paradigm: the agent-oriented programming paradigm.

Each programming paradigm promotes a particular point of view of programming. Using a specific view of programming as a programmer means taking a stance towards programming. Such a stance promotes the use of a set of concepts that structures the search for a program that is correct and solves the programming problem. For example, functional programming emphasises the concept of a function as the basic unit of computation, whereas object-oriented programming centres around concepts like that of an object and a class. These concepts provide a programmer with a set of tools to analyse and solve a particular software problem. The agent-oriented paradigm promotes a new view of programming that is based on taking an *anthropomorphic stance.* The concepts that play a central role in the agent-oriented paradigm thus are derived from our common sense explanations of human behaviour. A program is viewed as an intelligent agent and it is conceived of and analysed in terms of the common sense concepts of an *action*, a *belief*, a *goal*, and a *plan*. Where these same concepts in the human case are used to explain and predict the behaviour of humans, in analogy they are used here to explain and predict the behaviour of programs.

1.1 The Metaphor of Intelligent Agents

Intelligent agents mean many different things to different people. It is therefore appropriate to explain in somewhat more detail our use of agent terminology. In our definition of agent in the introduction, a central role is played by the word 'viewed'. An agent is any system that is *viewed* as such. In contrast, many researchers try to establish what the core content of an agent is. Definitions with lists of properties that aim at capturing the essence of an agent are prominent in the literature.

One can find many different types of properties that are associated with agents. These properties range from quite technical to less formal and more intuitive properties. To give an example of the former, mobility is a property that is often mentioned in the context of agents. Mobility of software is a quite technical issue that involves questions like: 'How can we realise mobile software?', and 'How can we guarantee security?', etc. An example of the latter category that is often used to define agents is the property of autonomy. It has been proposed to use the property of autonomy as a 'test' for agent-hood. The idea is that by measuring the degree of autonomy, the degree of agent-hood can be established. As far as the usefulness of concepts is concerned, autonomy is a concept that stands in sharp contrast with that of mobility. Whereas techniques for mobility have been developed by now and a clear sense of the problems has been gained, as far as autonomy is concerned little progress has been made. It is not clear how to realise autonomy in software, and since it has been difficult to define the concept itself, the concept has given rise more to confusion than to insight.

In between, a broad range of other properties are listed. For example, a property that is also often cited is that of adaptivity. An agent should have the capability of learning from past experience. Moreover, most researchers would agree that agents must be *social* agents that are able to interact and communicate. On top of that others insist that such agents must have *models* of other agents. The list of properties can easily be extended. The reader, however, may by now already have a sense of the problem. Since it is not obvious that each and every of these properties must be included in the list of defining properties of an agent, the following question remains. That is, how can we decide which properties should be included and which not? The only way to obtain an answer, it seems, is to try again to define the agent concept and we end up in a vicious circle.

The attempt to define agents by listing properties has still another effect. Even the short list of properties that we mentioned already gives the reader a feeling of high expectations. If agents can do all these things, they must truly be very powerful. A grand vision of the future then is easily born, but one still wonders how one should go about realising this vision. It is not at all clear what implications this list of agent properties has for the *development* of software systems.

Nevertheless, we believe there is much potential for agents. Our approach, however, will focus more on the need for enabling technology which provides for a platform for realising agent systems. In a sense, we turn things upside-down: Although agents may not *actually* possess any of the list of properties that we mentioned, we believe that by using agent terminology as a *conceptual tool* in the design of software a first step towards grander visions may nevertheless have been achieved.

In our view, it is not so important to provide a definite answer to the question 'What is an agent?'. As far as we are concerned, agents do not provide us with powerful new techniques that we can incorporate into our software systems, but agents do provide a very powerful and intriguing *metaphor* in the design of such systems. To put it crudely, just like one cannot point to any concrete 'objects' in a computer running an object-oriented program, one cannot point to a 'real' agent within a computer running agent software.

The view of agents as a useful and interesting new programming metaphor differs in important ways from other motivations that are put forward as reasons why agents are useful. In particular, we do not argue that agent systems are useful or even needed because of the *functionality* they provide. In this view, agent-based systems are supposed to *outperform* other types of software systems. We make no such claims about system performance. Our motivation is derived from the belief that the agent metaphor may enhance and improve the *design* of complex software systems.

To be of any use to computer scientists, however, we must make this metaphor more precise to be able to put it to good use. 'Intelligent agent' thus becomes a technical term in computing science. An agent is a conceptual tool which is developed in computing science with the purpose of solving the basic problem in programming:

> The basic problem in programming is the management of complexity. ... Program development should begin by focusing attention on the problem to be solved, postponing considerations of architecture and language constructs. (Chandy and Misra, 1988)

The agent-oriented programming paradigm aims to support this basic problem in programming by providing a framework in which all aspects of the design of a software system are analysed in terms of agent terminology. Agents as a modelling technique offer support in design, verification, and actual programming of software systems. In the design process, agent concepts are used to analyse, decompose and design a software system. These same concepts then can be used for the purpose of verification. Finally, these concepts can be used to develop an agent programming language. This is the main theme of this thesis. In this thesis, we will be preoccupied with the question what a programming language that is able to support agent design and construction looks like.

In our view, the agent paradigm provides a new *modelling technique*. The agent paradigm comes with its own *conceptual space* to model tasks, problems and solutions for programs. In this thesis, this conceptual space is confined to a minimum. The basic concepts included are that of a belief, goal, plan, and capability of a single agent. This single agent perspective is extended with communication features which allows the design of multi-agent systems. As we will show throughout the thesis, this minimal set of concepts already provides for a very rich framework. The use of the agent concept as a metaphor thus does not conceive of agents as being *really* intelligent or autonomous. If an agent is ascribed goal-directed behaviour or a pro-active attitude, for example, this should be taken in a metaphorical sense. The reader thus is encouraged to take the agent concept seriously as a *programming metaphor*.

From our point of view, agents are not so much "in the eye of the user", but instead become part of the training of a designer and programmer. Agent programming requires a particular design stance and in this sense it might be said that agents are "in the eye of the designer". It is interesting to note that a design stance and the claim that agents provide a solution to the basic programming problem yield quite different conclusions as far as the use of agent terminology is concerned than a typical AI stance. For example, whereas a well-known quote from McCarthy has it that such terminology is "most useful when applied to entities whose structure is very incompletely known", this is not at all in the spirit of the approach promoted here. In contrast, agent terminology is here used for the purpose of the design and *complete* specification of a software system.

1.2 Agent-Oriented Programming

Although agents are conceived of here primarily as a modelling technique, we believe that the success of this modelling technique depends on a supporting *agent programming framework*. This is a basic premise of our research and explains why this thesis is about an *agent programming language*. The starting point is, as Shoham puts it, that we must "make engineering sense out of these abstract concepts" (Shoham 1991).

There are several ways to make 'engineering sense' out of agent concepts. From the beginning, two approaches have dominated agent research. The first approach has focused on the design of agent logics for the specification and verification of such agents. This more theoretical work consists mainly of formal logics. Some examples are BDI logic, the logic of rational agency, and the KARO-framework (Rao 1996*b*, Singh 1994, Hoek et al. 1994, Linder et al. 1996, Cohen & Levesque 1990*a*). The main contributions of these formal approaches so far consist of a precise and formal analysis of agent concepts. Their use as specification languages, however, remains to be seen since so far no techniques have been provided for refining specifications in any of these languages to software systems. On the more practical side, a second approach has focused on the design of agent architectures. At another level of detail than the more theoretical research, the design of such architectures has led to many useful insights. At the same time, it is fair to say, we believe, that most of these agent architectures are quite complicated and in many cases offer too specialised solutions for building agents.

For these reasons, we have chosen in this thesis for a third, alternative approach: the design of an agent programming language. The advantages we see for an agent programming language are that such a language can integrate in a relatively simple way the basic agent concepts and at the same time offers a platform for building agent systems. This choice to study agent languages poses two central problems that we will address in this thesis. First, we need to answer the question what an agent language is. We will do that by way of construction and actually design two agent languages. Secondly, we need to address the relation with the other approaches towards agents. It has been especially difficult to establish a link between the theoretical agent logics and agent programming frameworks. In this thesis, we aim to provide such a link and make a concrete proposal to bridge the gap between theory and practice.

Several agent languages have been proposed in the literature (Dunin-Keplicz & Treur 1995, Poggi 1995, Rao 1996*a*, Shoham 1993, Thomas 1993). One of the first agent-oriented programming languages originated with the by now classic paper of Shoham on agent-oriented programming. The main contribution of this paper was to illustrate by means of a concrete proposal how a number of agent concepts can be supported and integrated in a programming framework. The paper, however, only provides an informal account of the language and does not provide a formal basis for an agent programming framework. The lack of a clear and formally defined semantics makes it difficult to formalise the design, specification and verification of programs. Other languages are based directly on logic (Giacomo et al. 1997, Lespérance et al. 1996, Fisher 1994, Wooldridge 1997). In general, it is very hard to integrate the minimal set of agent concepts outlined previously into such languages. For example, (Wooldridge 1997) includes the concept of knowledge but does not include a motivational concept like a desire or intention.

The need for a formal approach and the requirement that each of the basic agent concepts must be supported by the agent programming language motivates the design of two new agent languages in this thesis. Our approach is based on a combination of existing programming concepts from various well-known paradigms in order to model agent-oriented features. The clear advantage of this approach is that most of these concepts are well-understood, both from a theoretical and practical perspective. A second emphasis in our approach is on formal semantics. We think it is important to provide a formal semantics for a programming language. There are many reasons for a formal approach. But one of the more important ones is that a formal approach is the only viable approach that provides a basis for bridging the gap between agent theory and practice. Moreover, the formal semantics of a programming language allows a precise and formal comparison with some other related approaches.

Summarising, to promote this new style of agent-oriented programming, we think it is important to show at least three things. First of all, we believe it is important to show that the metaphor of intelligent agents enhances the programming and design task in particular domains by providing *convincing programming examples* which illustrate the power of using this metaphor (This is not a trivial task, cf. Kautz et al. (1994)). Some preliminary and suggestive work has been done to show this. Secondly, an *agent programming language* which supports the programming of agents should be provided. In this respect, it is particularly important to show how the agent language supports agent-oriented programming and to be clear about which aspects associated with agents are supported. For this purpose, an account of intelligent agents which explicates these notions in a precise and computationally useful way is required. Finally, we believe it is important to provide a *logic of agents* to reason about the agents written in the agent programming language. In this thesis, each of these issues is studied but our concern will be mainly with the design of an agent programming language.

1.3 Overview of Thesis

This thesis is divided into three parts. The first part is concerned with the design and introduction of an agent programming language that includes capabilities, beliefs, communication and planning. In this language, the concept of a goal and plan are more or less identified. This language is called 3APL (pronounced "triple-a-p-l"). 3APL is a new programming language which incorporates features from both imperative and logic programming and also includes some additional new features, which allow for an elegant description of many agent-oriented features. From imperative programming the language inherits the full range of regular programming constructs, including recursive procedures, and a notion of state-based computation. States of agents, however, are belief or knowledge bases, which are different from the usual variable assignments of imperative programming. From logic programming, the language inherits the proof as computation model as a basic means of computation for querying the belief base of an agent. These features are well-understood and provide a solid basis for a structured agent programming language. Moreover, 3APL agents use so-called *practical reasoning rules* which extend the familiar recursive rules of imperative programming in several ways. Practical reasoning rules can be used to monitor and revise the goals of an agent, and provide an agent with reflective capabilities.

In chapter 2, the programming language 3APL is informally introduced and its main features are discussed. This chapter introduces the single agent features and does not yet include communication. In chapter 3, the formal operational semantics of 3APL is defined. An operational semantics specifies the computation steps a program can perform. It thus describes the behaviour of a 3APL agent program. A special feature of 3APL are its so-called practical reasoning rules. These rules provide an agent with reflective capabilities and allow it to modify its current goals and plans. Since these rules are a distinguishing feature of 3APL, a separate chapter is devoted to this feature. In chapter 4, practical reasoning rules are studied in detail and some techniques associated with these rules are explained. Moreover, the semantics is extended and priorities on computation steps are introduced. In chapter 5, we then extend 3APL with communication. Different types of communication semantics between agents are explored and a number of alternatives are proposed. The main types of communication that are distinguished are that of communicating information and that of communicating a request. The first part closes with chapter 6 on meeting scheduling. An example 3APL program is presented that solves a multi-agent meeting scheduling problem.

In part II, the agent language 3APL is compared with a number of other agent languages from the literature. The emphasis is on a formal analysis and comparison to clarify the differences and similarities between these languages. In particular, the languages AGENT0 (Shoham 1993), AgentSpeak(L) (Rao 1996*a*), and ConGolog (Giacomo et al. 2000) are studied. In chapter 7, a formal semantics for AGENT0 is proposed. Since AGENT0 does not come with a formal semantics, we cannot formally relate AGENT0 and 3APL directly. However, the semantics that is developed in chapter 7 provides a solid basis for comparison with respect to the basic features of AGENT0 and 3APL.

In contrast, both AgentSpeak(L) and ConGolog are defined by means of a formal operational semantics. This provides a basis for rigorous and formal comparison and in chapter 8 we develop a methodology for such a comparison. The operational semantics is taken as a starting point to define a notion of (bi)simulation. Since an operational semantics defines the behaviour of a system, the framework for comparison is based on the idea of simulating the behaviour of agents from one language by agents from another language. By imposing a number of conditions on the type of simulation, we then obtain a measure for the expressivity of a programming language. The framework developed in chapter 8 is used in the next two chapters. In chapter 9, AgentSpeak(L) is compared with 3APL and it is shown that 3APL agents can simulate AgentSpeak(L) agents. In chapter 10, a comparison is made between ConGolog and 3APL. Although a number of differences exists between the two languages, by making two assumptions we are able to simulate ConGolog programs by means of 3APL agents.

In part III, the issue of bridging the gap between theory and practice is taken up again. In particular, this involves taking a closer look at the logics developed for agents. One of the most noticeable differences between agent logics and the agent programming languages studied in the first two parts concerns the concept of a goal. The motivational component in programming languages like AGENT0 and 3APL is more similar to a plan of action than a goal state that needs to be achieved. Such an achievement goal expresses a state of affairs that needs to be established by the agent and is quite different from a plan which is more like a recipe how to achieve such a state. It is therefore hard to formally relate BDI-like logics like (Rao 1996*b*) to agent programming frameworks like 3APL. To establish such a link, in part III a second agent language is designed. This agent language is called GOAL (for Goal Oriented Agent Language) and includes a so-called declarative goal that describes a state of affairs that must be achieved. Again, the language GOAL is designed by using familiar ideas from the programming language community. In particular, ideas underlying the programming language UNITY are taken as a starting point (cf. Chandy

& Misra (1988)). In chapter 11, the language GOAL is introduced and the main ideas incorporated in the semantics are explained. More specifically, the link between the beliefs and the goals of an agent are defined by a so-called commitment strategy. A commitment strategy determines if and when an agent may drop a goal. In chapter 12, we then show how the formal semantics for GOAL of the previous chapter can be used as a basis for a semantics of a temporal BDI-like logic for GOAL agents. The use of the logic is illustrated by proving a simple shopping agent correct.

Finally, in chapter 13 we summarise the main contributions of this thesis and make a number of suggestions for future work. From a more practical perspective, the agent programming language 3APL offers many roads for future research and provides a concrete framework for the construction of agents. From a more theoretical perspective, the agent language GOAL provides an interesting framework for more thorough research into agent concepts. In particular, the extension of GOAL to a multi-agent framework remains for future work. In the end, however, one might hope to combine both languages into a single agent language and combine the strong points of both languages into a single framework.

Part I:

The Agent Language 3APL

we have found that the flash of insight that sparks the creation of an algorithm is often based on operational, and even anthropomorphic, reasoning.

Chandy and Misra

CHAPTER 2

The Agent Programming Language 3APL

In this chapter, we make the agent concepts that were informally introduced in the introduction more precise and introduce the programming language 3APL. 3APL is an agent programming language and incorporates the basic notions like beliefs, goals and plans associated with an agent (cf. also Hindriks et al. (1998) and Hindriks et al. (1999*a*)). The syntax of the programming language is defined and the meaning of the language constructs is informally discussed.

The programming language 3APL combines a logical component with an operational component where the former provides agents with reasoning capabilities and the latter provides for an execution mechanism for plans. In fact, 3APL combines two different programming paradigms: *logic programming* and *imperative programming.* The usual states of imperative programming, however, are replaced with logical databases, whereas the assignment of imperative programming is replaced with updates on such databases. The shift from variable assignments from imperative programming to stores of information thus can be identified as one of the main differences between traditional imperative languages and the agent programming language 3APL. It also motivates a shift in terminology. Traditional states are now called *belief states* of the agent and traditional programs are now called *goals* of the agent. A 3APL agent is defined in these terms. An agent decides on the basis of its current beliefs and goals what to do. The action repertoire of an agent defines its *capabilities.* A set of so-called *practical reasoning rules* defines its planning capabilities.

2.1 Beliefs

The beliefs of an agent represent the information that the agent has about the task it must solve and its environment. Typical information about a task like, for example, meeting scheduling, is that a meeting has five relevant parameters:

an identifier to denote the subject of the meeting, attendants, date/time, length and location. The environment typically consists of other agents and an agent thus may have beliefs about other agents. For example, agent A may believe that agent B has some meeting at 5pm.

There are many aspects to choosing a knowledge representation language, and the study of such choices is the area of *knowledge representation* (Oérez & Benjamins 1999). In principle, the programming language 3APL can be combined with any type of knowledge representation language. It can be a modal language, a first order language, or any other language that fits the construction of particular types of agents best. The knowledge representation language thus is a *plug-in* feature of 3APL and the choice of such a language can be decided upon during the design of an agent system. The only constraint on the representation language is that it should come with a clear inference mechanism for deriving facts from stored facts and associated update mechanisms for updating stored facts. Moreover, a predicate-argument structure for basic facts is assumed in this thesis.

This thesis is not about knowledge representation, however. Both for technical reasons and for ease of exposition, we therefore will select a particular knowledge language that serves our purposes. The only requirement that constrains our choice is that the language should be a general purpose knowledge representation language that comes with the associated machinery for updating and inspecting the beliefs of an agent. In this thesis, we have chosen *first order languages* as the agent knowledge representation language. First order languages can naturally model many types of information, are well-known and have a lot of expressive power. There are many good introductions to first order logic, for example, (Mendelson 1979, Enderton 1972). The consequences of choosing some other formalism for representing knowledge are not explored in this thesis.

The beliefs of an agent thus are formulas from some first order language $\mathcal{L}$. A first order language $\mathcal{L}$ is built from a signature $\Sigma = \langle \mathsf{Pred}, \mathsf{Func} \rangle$ of predicate symbols Pred and function symbols Func with associated arities, and a countably infinite set of variables Var with typical elements $x, y, z, \ldots, x_1, \ldots$. Terms are built from Func and Var as usual and we use Term to denote the set of all terms.

Definition 2.1.1 *(language of beliefs)*
The set of first order formulas $\mathcal{L}$ is inductively defined by

- if $p \in \mathsf{Pred}$ of arity n and $t_1, \ldots, t_n \in \mathsf{Term}$, then $p(t_1, \ldots, t_n) \in \mathcal{L}$,
- if $t_1, t_2 \in \mathsf{Term}$, then $t_1 = t_2 \in \mathcal{L}$,
- if $\varphi \in \mathcal{L}$, then $\neg\varphi \in \mathcal{L}$,
- if $\varphi \in \mathcal{L}$ and $\psi \in \mathcal{L}$, then $\varphi \wedge \psi \in \mathcal{L}$,
- if $\varphi \in \mathcal{L}$ and $x \in \mathsf{Var}$, then $\forall\, x(\varphi) \in \mathcal{L}$.

Other logical constructs like disjunction $\vee$, implication $\rightarrow$, and existential quantification $\exists$ are defined as the usual abbreviations. A number of special syntactic classes and concepts are associated with first order languages. First, a formula of the form $p(\vec{t})$ is called an *atom*; the set of atoms is denoted by At. The notions of *free* and *bound variable* are defined as usual (cf. Lloyd (1987)). A variable x is bound if it occurs within the scope of some quantifier $\forall\, x$, otherwise it is free. The set of free variables in an expression e is denoted by $\mathsf{Free}(e)$. A *ground atom* is an atom without occurrences of free variables; a *ground term* is a term without occurrences of free variables. A formula without free variables is also called a *sentence*. The usual *entailment relation* for first order logic is denoted by $\models$ (cf. Mendelson (1979)). Informally, $T \models \varphi$ if φ is implied by the set of sentences T.

The beliefs are used by the agent to choose an appropriate plan to achieve its goals in the given circumstances, to make choices left open in a plan, to decide whether or not to revise a currently adopted plan, and, last but not least, to store (and retrieve) information that will be useful in the future life of the agent. The storage, retrieval and manipulation of information in databases has traditionally been the concern of *database theory* (Minker 1988), and of AI research areas like *knowledge-based systems* and *expert systems* (Stefik 1995). Therefore, it should not come as a surprise that there are numerous links between intelligent agent programming and these other fields of expertise.

Now we are in a position to formally define the concept of a *belief base* of an agent. A belief base is defined in terms of the underlying knowledge representation language, in our case a first order language.

Definition 2.1.2 *(belief base)*
A *belief base* σ is a consistent set of sentences $\sigma \subseteq \mathcal{L}$, i.e. $\sigma \not\models \mathsf{false}$.

Note that a belief base is a set of closed formula, i.e. it does not contain any free variables. Operations to *inspect or query* and *modify* a belief base are introduced below.

2.2 Personal Digital Assistants

One of the more promising applications of intelligent agents are so-called *personal digital assistants* (PDAs). A PDA is a computer-based office automation system that acts on the user's behalf (Maes 1994, Lennon & Vermeer 1995). The user can delegate specific tasks to such agents. Typically, the tasks handled by personal assistants are supposed to be repetitive, boring and time-consuming tasks (Hoyle & Lueg 1997, Maes 1994, Lennon & Vermeer 1995).[1] Tasks like for example news filtering and meeting scheduling are especially amenable to automatisation by means of personalised agents. Such tasks are user-specific in the sense that each user has its own particular preferences of how to perform the

[1]The suggestion that we need so-called *intelligent* agents for these tasks also suggests that we need not take this characteristic of intelligence too seriously.

task. This characteristic is a necessary prerequisite for the use of the *personalised* agents metaphor instead of using more general preprogrammed strategies. Secondly, these tasks have in common their repetitive nature which allows such agents to *learn* how to perform a task by observing the user. Of course, it is also possible that the user instructs the agent itself and informs the agent directly of his/her preferences.

Another interesting aspect of the personal assistant metaphor is that it requires an analysis of the user - both through building a *personal profile* by the agent software (user modelling) as well as by the agent designer. This type of model building is necessary for the PDA to be both *competent* as well as *trustworthy.* The metaphor has been explained as a shift away from the *direct manipulation* metaphor to the *indirect management* metaphor (Maes 1994). Whereas we were used to a state of affairs where we had to tell computer programs step by step what to do, nowadays we require computer systems to take more initiative and perform certain routine tasks automaticly without our own intervention. The personal assistant metaphor thus has given rise to a new approach to user interface design, and is very useful also in the area of computer-supported cooperative work (Baecker 1993).

Throughout this thesis, examples of personal assistants are used to illustrate features of the agent programming languages that are introduced. That is, pieces of code that may be part of a personal assistant program are provided. To illustrate the agent programming language 3APL, a personal assistant that serves as a time management system is used (cf. Jennings & Jackson (1995)). This type of personal assistant supports their user with the scheduling of activities. They may also provide their users with related information like, for example, information about transportation from one location to another. In chapter 6, a more complete example of a meeting scheduling system is provided.

Example 2.2.1 In the examples, we use the Prolog-style convention that strings starting with capital letters are variables. The beliefs of personal assistants in this example concern the agenda of its user. A predicate *meet* with five arguments *Ident*, *Time*, *Length*, *Location* and *Attendants*, respectively, is used to keep track of the items in the user's agenda. For example, the sentence $meet(IA\_course, 11:00, 2:00, c009, user)$ states that at 11am the user has to teach a course on Intelligent Agents for 2 hours in room $c009$. An integrity constraint associated with the predicate *meet* states that items in the agenda are not allowed to overlap in time. Formally, this is represented by:[2]

$$\begin{aligned}&(meet(Id1, T1, Len1, Loc1, A1) \wedge meet(Id2, T2, Len2, Loc2, A2) \wedge \\ &\quad T1 \leq T2 < T1 + Len1) \\ &\qquad \rightarrow Id1 = Id2 \wedge T1 = T2 \wedge Len1 = Len2 \wedge Loc1 = Loc2 \wedge A1 = A2\end{aligned}$$

The predicate $location(Loc, Time)$ is used to keep track of the user's location at a particular time. The predicate is assumed to be persistent. That is, if there is no information to the contrary the location of the user at a time t' such

[2]Throughout this thesis, free first order variables in the examples concerning belief bases are implicitly universally quantified.

that $t < t'$ is by default assumed to be the same as that of time t. Since the information concerning the location of the user is derived from the activities and associated locations in the user's agenda, a closed world assumption associated with the *meet* predicate can be used to implement this. In that case, we can specify the relation between the two predicates as follows:

$$\begin{aligned}&(meet(Id1, T1, Len1, Loc1, A1) \wedge T1 \leq Time \wedge A \in A1 \wedge \\ &(T1 < T2 < Time \rightarrow \neg(meet(Id2, T2, Len2, Loc2, A2) \wedge A \in A2))) \rightarrow \\ &\qquad location(A, Loc, Time)\end{aligned}$$

2.3 Actions and Agent Capabilities

To modify its environment, an agent is equipped with a set of *basic actions.* Basic actions are one of the types of basic goals in the agent programming language. They specify the capabilities that an agent has to achieve a particular state of affairs.

It is important in this context to emphasise that our view of an agent is motivated by the metaphor based on common sense notions like belief and goal and an agent is viewed as a mental entity. Consequently, actions are conceived of as mental actions transforming the mental state of the agent. Although it may well be that an agent changes its environment through some interface which depends on the execution of basic actions in the agent language, there is nothing in the agent language which requires such an interface with an environment external to the agent. The use of intelligent agents to control robots is an example where such an interface is required. Personal assistants, like agents that manage the agenda of their user, however, do not require any interface to control some external environment. Note, however, that in both cases the beliefs of an agent represent something external to that agent itself. Blocks and other objects in the first case, activities, locations and so on, in the second case.

Because a 3APL agent is a mental entity, from the point of view of the programming language, basic actions are actions which affect the mental state of the agent. Basic actions therefore are defined as updates on the beliefs of the agent. This choice reflects the fact that the beliefs of the agent represent the environment which may change due to the performance of actions. In correspondence with such changes, the beliefs of the agent are updated.

Definition 2.3.1 *(basic actions)*
Let Asym be a set of action symbols with typical elements $\mathsf{a}, \mathsf{a}', \ldots, \mathsf{b}, \ldots$ each with a given arity. Then $\mathsf{a}(t_1, \ldots, t_n)$ is a *basic action* for any action symbol $\mathsf{a} \in \mathsf{Asym}$ and terms $t_1, \ldots, t_n \in \mathsf{Term}$ where n is the arity of a. The set of basic actions is denoted by Bact.

Example 2.3.2 The personal assistant of our running example is capable of performing a number of basic actions related to managing an agenda and informing the user about appointments etc.

The action $\mathsf{ins}(meet(Ident, Time, Len, Loc, Att))$ inserts in the agenda that is maintained in the belief base the item *Ident* scheduled at time *Time* in case it does not violate any integrity constraints associated with the agenda. This action enables the agent to keep the agenda up to date.

To communicate with the user, the action

$$\mathsf{inform\_user}(String, Ident, Time, Len, Loc, Att)$$

is introduced. This action allows the agent to inform the user about an item that is scheduled in its agenda. *String* denotes a string which is prefixed to the output of the parameters concerning the activity.

In general, actions can only be performed if certain conditions hold. These conditions are called the *preconditions* of the action. In the previous example, the precondition of action ins included a consistency constraint. Only if the item to be inserted does not give rise to conflicts with other scheduled activities is it possible to insert such a new item. In case an action is executed, it has certain effects. Such effects may be captured by yet another condition called the *postcondition* of the action. The ins action results in a state where the item is inserted into the agenda. It is possible to formally characterise the pre- and postconditions, but we do not discuss such formal techniques here. In chapter 10, we will see one example of a technique to specify such conditions based upon the *situation calculus*.

2.4 An Abstract Programming Language

3APL is an *abstract* programming language, in the sense that it does not include any particular set of concrete actions nor does it dictate the use of a particular knowledge representation formalism. The choice of the most suitable basic actions and the choice of the most suitable knowledge representation formalism need to be settled at design time. In fact, in a multi-agent system each agent can be equipped with its own set of capabilities and its own expertise on a particular subject. Even within single agents it is conceivable to use different formalisms to code the beliefs of an agent or to define the pre- and postconditions of actions. This introduces interaction and translation problems, however, between and within agents (see for a treatment of such issues (Eijk et al. 1998)). 3APL thus provides for a *framework* for programming agents, and is best conceived of as a kind of *coordination language* (Papadopoulos & Arbab 1998) in the sense that it is a language for coordinating heterogeneous activities and knowledge representation languages of agents.

The philosophy behind our approach is that agent programming does not aim so much at *replacing* other programming formalisms with a more suitable alternative, but instead should be viewed as offering a high-level framework to *integrate* different services programmed in different programming paradigms. Agents do not replace databases, information retrieval techniques, etc. but rather facilitate the integration of such diverse systems and techniques in a coherent fashion.

This view of agents already provides for a rudimentary software engineering methodology. The core of this methodology is the set of agent concepts. The agent terminology provides conceptual tools that allow the description of software systems at a high level of abstraction. The agent concepts moreover are both intuitive and natural to use. Because of their abstract nature, agent concepts support the design of complex software systems. For the same reason, however, agent terminology is not universally applicable. Agent concepts, for example, are not very suitable for things like solving sorting problems, solving equations, etc. Prime examples where agent concepts are useful concern business applications and personal assistant applications.

2.5 Goals and Plans

In common sense, and in most logics of agents, the goals of an agent describe the state of affairs an agent would like to achieve. These types of goals are *declarative or descriptive* in nature, and are also called *goals-to-be*. Declarative goals do not specify *how* the agent should go about realising such goals. There is, however, yet another notion of a goal which does specify a plan of action. Such goals provide for a *procedural* perspective and do not focus so much on the resulting state of affairs. This second type of goal is also called a *goal-to-do*, because it specifies a recipe for action that the agent intends to follow. Goals-to-do are similar to the motivational concept of intention, which has been analysed by Bratman in terms of plans (Bratman 1987). They represent the *adopted plans* of an agent to achieve a particular state.

First, our concern will be to introduce an agent programming language which incorporates a procedural perspective on goals. In a later chapter, we will then return to the issue of incorporating declarative goals into an agent language. From a computational perspective, it is quite natural to focus first on the procedural type of goal because this notion can be analysed in terms of plans. In Artificial Intelligence, moreover, the concept of a plan has since long been recognised as similar to that of an imperative program. Goals in this sense thus can be viewed as a kind of imperative program. In this chapter, therefore, if we simply write 'goal' we mean the procedural type of goal or a plan of action.

2.5.1 Basic Goals

Goals - like imperative programs - are structures built from a set of programming constructs and a set of so-called *basic goals*. The programming constructs are used to compose such basic goals, for example, into a sequential order. As explained previously, the main difference with imperative programming, however, is that we have replaced imperative assignments with information updates. The set of basic goals consists of the most simple goals and is different from the basic constructs in imperative programming.

There are three types of basic goals. First, a basic action is a basic goal. A basic action is one of the building blocks for constructing a plan and allows the agent to adopt a plan to modify its current beliefs. Basic action goals do not allow the agent to introspect its beliefs.

A second type of basic goal is a *test goal* $\varphi?$ where φ is a formula from the knowledge representation language. A test goal expresses a condition on the beliefs of the agent. It allows an agent to *introspect* its beliefs and is evaluated relative to the current beliefs of the agent. A goal $\varphi?$ first of all allows an agent to find out whether or not it believes φ, that is, whether or not its current beliefs entail the formula φ. Actually, however, things are slightly more complicated because φ may contain free variables and does not have to be a sentence. The use of free variables in test goals is that it provides for a mechanism to compute correct instances or bindings for these variables. As in logic programming, a test thus may be used to derive instances for free variables. Such instances must be correct in the sense that they must be entailed by the beliefs of the agent. Because a test binds a value to a variable, it is somewhat similar to an assignment in imperative programming. The main differences are that a test goal is evaluated by means of logical proof (as in logic programming), and a test can only be used to *initialise* a variable to some value, not to update the value assigned to a variable.

Example 2.5.1 The test $\mathit{location}(\mathit{user}, \mathit{Loc}, \mathit{Time})?$ can be used to retrieve the location of the user at a given time (or, in case a location would already have been bound to the variable *Loc*, to retrieve the times the user was or is going to be at that particular location according to its agenda).

The last type of basic goal is an *achievement goal*. An achievement goal simply is an atom from the knowledge representation language $\mathcal{L}$ of the form $p(\vec{t})$. An achievement goal provides for an abstraction mechanism similar to that of a procedure call in imperative programming. These goals thus are not themselves plans of actions but must be replaced with an appropriate plan to achieve a particular goal coded by the atom $p(\vec{t})$.

The programming language does not enforce any semantic relation between an achievement goal $p(\vec{t})$ and the belief atom $p(\vec{t})$. The meaning of an atom $p(\vec{t})$ in the belief base and an occurrence of that same atom in the goal base thus are not formally related. For example, if an agent believes that $p(\vec{t})$ is the case, it still may have the achievement goal $p(\vec{t})$ too. It is left to the programmer to see to it that both uses of the same atom are coherent. The use of atoms as achievement goals thus is very different from the use of atoms as beliefs. Whereas in the latter case atoms are used to represent and therefore are of a declarative nature, in the former case they have a procedural meaning.

Example 2.5.2 An example of an achievement goal is

$$\mathit{schedule}(\mathit{Ident}, \mathit{Time}, \mathit{Len}, \mathit{Loc}, \mathit{Att})$$

which may be used for scheduling an activity *Ident*. A plan to implement this goal is given below by a practical reasoning rule.

2.5.2 Composed Goals

Composed goals are built from basic goals by means of the programming constructs for *sequential composition*, *nondeterministic choice* and *parallel composition* which are well-known programming constructs from imperative programming. Sequential composition allows the specification of a sequence of subgoals $\pi_1;\ \pi_2$ which is a plan to first do π_1 and after finishing π_1 continue with plan π_2. Nondeterministic choice is a construct that allows to specify disjunctive goals $\pi_1 + \pi_2$ which is a plan to do either π_1 or π_2. A goal $\pi_1 + \pi_2$ is also called a *choice goal.* An agent also may have goals that are executed in parallel. Such *parallel goals* are denoted by $\pi_1 \| \pi_2$. Apart from such explicit parallelism, an agent may also have implicit parallel goals since it may adopt more than one goal simultaneously.

Definition 2.5.3 *(goals)*
The set of *goals* Goal is inductively defined by:

- $\mathsf{Bact} \subseteq \mathsf{Goal}$,
- $\mathsf{At} \subseteq \mathsf{Goal}$,
- If $\varphi \in \mathcal{L}$, then $\varphi? \in \mathsf{Goal}$,
- If $\pi_1, \pi_2 \in \mathsf{Goal}$, then $(\pi_1;\ \pi_2), (\pi_1 + \pi_2), (\pi_1 \| \pi_2) \in \mathsf{Goal}$.

Example 2.5.4 The goal

$$location(user, Location, 12:00)?;$$
$$\mathsf{ins}(meet(lunch, 12:00, 1:00, Location, user))$$

is an example of a sequential goal composed of a test and a basic action. An important aspect of the parameter mechanism of 3APL is how different occurrences of the same variable in a goal relate to each other. The rule is that the first occurrence of a free variable in a sequential composition *implicitly* binds all later occurrences of that same variable.

Thus, by executing the test in our example goal, a binding for the variable *Location* is derived from the current beliefs that denotes the location of the user at 12am. Let's assume that the location retrieved is the *office* of the user. Since the second occurrence of the variable *Location* is implicitly bound by the first occurrence, the value computed by the test for this variable is then used to instantiate the second occurrence and a lunch at 12am in the office of the user is inserted in the agenda.

We can define new, useful programming constructs in terms of the programming constructs introduced so far. By using both test goals and nondeterministic choice, for example, we can define a new construct IF..THEN..ELSE.. as follows:

$$\mathsf{IF}\ \varphi\ \mathsf{THEN}\ \pi_1\ \mathsf{ELSE}\ \pi_2 \overset{df}{=} (\varphi?;\ \pi_1 + \neg\varphi?;\ \pi_2)$$

This conditional programming construct continues with executing π_1 in case (an instance of) φ is entailed by the beliefs of the agent and will execute π_2 in case $\neg\varphi$ is entailed. An important difference with similar conditional constructs in other languages, is that sometimes neither branch of IF φ THEN π_1 ELSE π_2 is executed. This happens when neither (an instance of) φ nor (an instance of) $\neg\varphi$ is entailed by the current beliefs of an agent. Since tests are introspective actions into the beliefs of the agent, sometimes the agent does not have enough information to decide a test and the conditional action as defined above may result in a *deadlock*, i.e. it is not possible to continue execution at this point in the program. Another type of operation that was not introduced as part of the programming language is an *iteration operator*. The language does not include a construct like the traditional WHILE..DO... Iteration, however, can be defined in terms of recursive rules which are introduced below.

As is clear from the discussion so far, goals operate on the belief base of an agent. They both change and derive information from the agent's beliefs and their main purpose is in manipulating the beliefs of the agent. 3APL goals, therefore, can be viewed as *belief update operators*.

Remark 2.5.5 *(note on terminology)*

1. The notion of a *test goal* as well as that of an *achievement goal* was first introduced in (Rao 1996*a*).

2. The notion of a *sub*goal refers to a part of a larger goal. For example, π_1 is a subgoal of $\pi_1;\ \pi_2$. The concept of a subgoal can formally be defined by means of the concept of a context that is defined in chapter 4.

3. Note that in the agent language the common sense notions of (adopted) plans and goals are both represented by expressions from the set of goals Goal. Thus, the distinction between these two notions is not formally represented in the language, but can only be made using our common sense understanding of these notions. In the sequel, we will use either one of these two notions dependent on which of the two is the more natural one to use in a given context. Informally, a plan also refers to a sequence of actions.

4. We use the notion of a *goal* rather than that of *intention*. The reason is that the notion of a goal is a more general notion than that of intention. Intentions are usually viewed as some kind of *choice* with an associated level of *commitment* made to that choice ((Cohen & Levesque 1990*a*, Bratman 1987)). The commitment made to a choice determines when an agent will reconsider or drop its intention. An agent may adopt several commitment strategies towards its intentions. In the programming language, a goal does reflect a choice the agent has made. However, there is no explicit level of commitment associated with each of the goals of an agent. The commitment strategies or revision strategies of an agent are more or less implicit in the practical reasoning rules of an agent because these rules

determine when a goal may be modified. The practical reasoning rules of an agent thus determine whether or not a goal can be considered as an intention.

2.6 Practical Reasoning Rules

To achieve its goals and to monitor its plans, a 3APL agent uses *practical reasoning rules.* Practical reasoning rules supply the agent with a facility to manipulate its goals. Whereas goals operate on the beliefs of an agent, rules operate on the goals. Practical reasoning rules can be used to build a plan library from which an agent can retrieve plans for achieving an achievement goal $p(\vec{t})$. They also provide the means to revise and monitor goals of the agent.

The name *practical reasoning rules* derives from the role they play in the operation of intelligent agents. They supply agents with reasoning capabilities to reflect on their goals. The application of practical reasoning rules is similar in certain respects to common sense practical inference of plans to achieve goals as used by human agents. Informally, this type of common sense reasoning can be paraphrased as follows. From a goal to achieve φ and the belief that the plan π is sufficient to achieve φ, the agent concludes that it is reasonable to adopt the plan π. This type of reasoning is also called *means-end reasoning.* The goals considered in this type of reasoning are declarative, and only simple achievement goals are similar to this type of goals in 3APL. Note that in our explanation of practical reasoning the conclusion of this type of reasoning is the adoption of a new goal (and not the performance of some action, as some philosophers claim that the result of practical reasoning is).

Apart from means-end reasoning, an agent may also have to reconsider the plans it adopted. The reasoning involved in reconsidering one's goals follows a similar pattern as the means-end reasoning in the previous paragraph. Informally, a commitment to a plan π and a belief that the situation requires replacing this plan with an alternative course of action π', provide the agent with reasons to conclude that it should adopt π' and remove π from its current goals. We will classify both types of reasoning under the heading of *practical reasoning*, and illustrate below how rules are used to implement this type of reasoning.

In the example patterns of practical reasoning we presented, it is not too difficult to recognise a type of conditional rule. The practical reasoning rules we are about to introduce capture the essence of this pattern of reasoning. Presented with a plan and a belief concerning the current situation, a practical reasoning rule allows the agent to adopt a new plan which replaces its original plan. Practical reasoning rules thus code the know-how of an agent.

Because monitoring of plans is a meta-activity unlike the matching of plans with achievement goals, its technical realisation requires an extension to the language with a facility to refer to plans themselves. For this purpose, we introduce so-called *goal variables* which range over the goals or plans of an agent. We extend the notion of goals and introduce a new concept of *semi-goals* which are constructed from basic actions, achievement goals, tests, sequential composition, nondeterministic choice and parallel composition like regular goals, but which may also include goal variables. The goal variables in semi-goals should be thought of as *place-holders* for goals. Also, the notion of a binding is now extended to include the binding of goals to goal variables.

Definition 2.6.1 *(semi-goals)*
Let Gvar be a countably infinite set of goal variables (ranging over goals) with typical elements X, X' such that $\mathsf{Gvar} \cap \mathsf{Var} = \varnothing$.
The set of *semi-goals* SGoal is inductively defined by the syntactic rules from definition 2.5.3, where Goal is replaced by SGoal, and the rule

- $\mathsf{Gvar} \subseteq \mathsf{SGoal}$.

The set of practical reasoning rules is built from semi-goals and first order formulas from the knowledge representation language $\mathcal{L}$. We introduce three different types of practical reasoning rules.

Definition 2.6.2 *(practical reasoning rules)*
Let $\pi_h, \pi_b \in \mathsf{SGoal}$ be semi-goals, and $\varphi \in \mathcal{L}$ be a first order formula. Then the set of *practical reasoning rules* PRule is defined by:

- $\pi_h \leftarrow \varphi \mid \pi_b \in \mathsf{PRule}$, such that any goal variable X occurring in π_b also occurs in π_h,
- $\leftarrow \varphi \mid \pi_b \in \mathsf{PRule}$, where $\pi_b \in \mathsf{Goal}$, and
- $\pi_h \leftarrow \varphi \in \mathsf{PRule}$.

Informally, one can read a practical reasoning rule $\pi \leftarrow \varphi \mid \pi'$ as stating that if the agent has adopted some goal or plan π and believes that φ is the case, then it may consider adopting goal π' as a new goal. The application of a rule is based on pattern-matching: A rule $\pi_h \leftarrow \varphi \mid \pi_b$ applies to a current (sub)goal π of an agent if π matches with π_h. The guard of the rule (more precisely, an instance of the guard) must also be entailed by the agent's current beliefs for a rule to be applicable. A current (sub)goal of an agent that matches with the head π_h of a rule is *replaced* by the body of that rule when the rule is applied. Practical reasoning rules are best conceived of as extending recursive procedures from imperative programming. They extend recursive rules because they can also be used to modify plans in arbitrary ways, as we illustrate below, which is not a feature of recursive procedures.

A practical reasoning rule has the following components:

Definition 2.6.3 *(head, body, guard, global and local variables)*
Let $\pi \leftarrow \varphi \mid \pi'$ be a practical reasoning rule.

- π is called the *head* of the rule,
- π' is called the *body* of the rule,

- φ is called the *guard* of the rule,
- the free first order variables in the head of a rule are called the *global variables* of the rule,
- all other first order variables in the body of a rule which are not global are called *local variables.*

By convention, we write $\pi_h \leftarrow \pi_b$ for $\pi_h \leftarrow \mathsf{true} \mid \pi_b$. Rules of the form $\leftarrow \varphi \mid \pi$ are said to have an *empty head*, whereas rules of the form $\pi \leftarrow \varphi$ are said to have an *empty body.* Rules with an empty body are used to *drop goals.* Rules with an empty head are used to *create new goals* independent of the current goals of an agent.

The distinction between global and local variables is made to separate the local data-processing in the body by means of the local variables which are only used in the body, from the global variables which may be used both as input variables and output variables. Global variables can be used as input variables to supply data to the body of the rule, and as output variables to return computed results. When a rule is applied, all its local variables in the body are renamed to variables which do not occur anywhere else in the goals of the agent. This renaming provides for an *implicit scoping mechanism*, similar to that in logic programming. Note that values that are computed for variables that only occur in the guard of a rule are not used at all in the remainder of a computation.

Example 2.6.4 We first give an example of a *plan rule* which is used to find a matching plan for the achievement goal $schedule(Ident, Time, Len, Loc, Att)$ of example 2.5.2 and illustrate the application of this rule to a particular instance of this goal.

$$\begin{array}{l}
schedule(Ident, Time, Len, Loc, Att) \\
\leftarrow location(user, FromLoc, Time) \wedge user \in Att \mid \\
\quad transport(Means, FromLoc, Loc, DurTrans)?; \\
\quad \mathsf{ins}(meet(Means, Time - DurTrans, DurTrans, FromLoc, user)); \\
\quad \mathsf{ins}(meet(Ident, Time, Len, Loc, Att))
\end{array}$$

The plan in the body of this rule is to schedule the activity *Ident* by inserting it in the agenda at the appropriate time. But the agent also supports the user by automatically calculating which means of transportation are sufficient to get to the location of the activity in time and reserving time for travelling in the agenda by means of the first ins action in the plan. This is achieved by retrieving this information by means of the test $transport(Means, Fromloc, Loc, DurTrans)$? where *Means* returns the type of transportation required to get from *Fromloc* to the destination *Loc* and *DurTrans* denotes the time it takes to get to the destination.

Now suppose that the agent has a goal of scheduling a course on Intelligent Agents in Utrecht at 11am of one hour:

$$schedule(IA\_course, 11:00, 1:00, utrecht, students)$$

Moreover, also assume that the agent believes that:

$$\begin{array}{l}
(meet(Id, T, Len, Loc, A) \leftrightarrow \\
\quad (Id = meeting \wedge T = 9:00 \wedge Len = 1:00 \wedge Loc = amsterdam \wedge \\
\quad A = \{user, mark\})) \wedge \\
transport(train, amsterdam, utrecht, 0:45) \wedge \\
((meet(Id1, T1, Len1, Loc1, Att1) \wedge T1 \leq Time \wedge \\
(T1 < T2 < Time \rightarrow \neg meet(Id2, T2, Len2, Loc2, Att2))) \rightarrow \\
\qquad\qquad location(Loc1, Time)) \wedge \\
user \in students
\end{array}$$

In that case, the rule for scheduling an activity is applicable. First, the head of the rule matches with the current achievement goal of the agent; the substitution $\{Ident = IA\_course, Time = 11:00, Len = 1:00, Loc = utrecht, Att = students\}$ is a (most general) unifier of the head of the rule and the current goal of the agent. Second, because of the closed world assumption associated with the predicate *meet*, the agent can derive $location(user, amsterdam, 11:00)$ from its beliefs, which is an instance of the guard of the rule.

All the global variables in the head of the rule are used as input variables in this example. These input values are used to instantiate the body of the rule. By replacing its current achievement goal with the new plan the agent ends up with the new goal:

$$\begin{array}{l}
transport(Means, amsterdam, utrecht, DurTrans)?; \\
\mathsf{ins}(meet(Means, 11:00 - DurTrans, DurTrans, amsterdam, \{user\})); \\
\mathsf{ins}(meet(IA\_course, 11:00, 1:00, utrecht, students))
\end{array}$$

Note that both the global input variables and the variable *FromLoc* that occurs in the guard are instantiated in the body.

The previous example illustrates the use of a simple rule of the form $p(\vec{t}) \leftarrow \varphi \mid \pi$ where $p(\vec{t})$ is an achievement goal. This type of rule is similar to a (recursive) procedure in imperative programming. These rules are useful for specifying a *plan* to achieve an achievement goal and are also called *plan rules.* A plan rule encodes the *procedural knowledge* of an agent. A set of such rules constitutes a plan library which an agent can consult to find plans to achieve its goals.

Example 2.6.5 We provide two more rules to illustrate the use of recursion. The purpose of these rules is simply to provide the user with a pre-calculated list of items in its agenda. Each of these items consists of the activity involved, and the time, the duration, the location and the attendants of the activity.

$$\begin{array}{l}
inform\_list(MeetList) \leftarrow MeetList = [[Ident, Time, Len, Loc, Att], List] \mid \\
\quad \mathsf{inform\_user}(\texttt{scheduled}:, Ident, Time, Len, Loc, Att); \\
\quad inform\_list(List) \\
inform\_list(MeetList) \leftarrow MeetList = [\,] \mid
\end{array}$$

The rules recursively inform the user of the consecutive items in the list *MeetList*. Note how the second rule which has an empty body is used to deal with the termination of the recursion. The two rules also illustrate how recursion can be used to program iteration.

Apart from the plan rules illustrated so far, there are a number of other types of rules. As we already mentioned, rules with an empty head $\leftarrow \phi \mid p(\vec{t})$ can be used for *goal creation*. Such rules are useful for implementing reactive behaviour of an agent; moreover, rules of the form $\pi \leftarrow \phi$ can be used to *drop* goals.

The use of the goal variables resides in the way they support very general modification of goals. The use of these variables in practical reasoning rules allows for all kinds of revision and monitoring facilities. Practical reasoning rules, therefore, provide an agent with *reflective capabilities* concerning its goals. In particular, they may be used to deal with failure of a current plan of an agent.

Example 2.6.6 In this example we provide a very simple illustration of the use of other practical reasoning rules than plan rules and of goal variables. In chapter 4, we will study in detail the use and expressive power of these rules.

Recall the plan from example 2.6.4 for scheduling a meeting *Ident*. This plan first attempts to compute the appropriate means of transportation to get to the location of the meeting *Ident* which the agent tries to schedule, then attempts to insert the activity of travelling and the time it takes in the agenda in the appropriate place, and finally attempts to schedule the meeting *Ident* itself.

Of course, this plan will not always succeed. In particular, one case in which it will fail is when the time of the meeting that is to be scheduled conflicts with the time of another meeting that already has been scheduled. Now, in 3APL there are two ways of dealing with this failure. On the one hand, there is the traditional approach which insists that the plan is *incorrect* and has to be *rewritten* to correct the errors in it. On the other hand, in 3APL, there is a second option to deal with this failure by using practical reasoning rules. In that case the original plan does not have to be rewritten, and can be viewed as the correct plan *in the normal case*. Rules to deal with the failure are added to deal with the *abnormal cases* in which the plan would fail.

A practical reasoning rule to deal with the case in which a meeting cannot be scheduled due to conflicts with already scheduled meetings is presented next. The guard of this rule implements the condition that the meeting that needs to be scheduled conflicts with another meeting in the agenda of the user. The crucial parameter here which needs to be retrieved from the scheduling plan is the time at which the meeting is supposed to begin. This is retrieved from the scheduling plan by matching the head of the rule which consists of a goal variable and an ins action with the scheduling plan. The variable *Time* then retrieves the time the meeting is supposed to begin. In the guard, it is checked whether this time conflicts with another meeting which has already been scheduled. If that is the case, then the rule is applicable and the original scheduling plan is replaced by an action to inform the user of an unsuccessful attempt to schedule the meeting. In the rule, the goal variable is used to match with the first part of the scheduling plan which consists of a test and the ins action to schedule travelling time. Of course, in case the meeting cannot be scheduled these actions also need to be removed.

$$
\begin{array}{l}
X;\ \mathsf{ins}(meet(Ident, Time, Len, Loc, Att)) \\
\leftarrow meet(Id', T', Len', Loc', Att') \wedge \\
\quad ((Time \leq T' < Time + Len) \vee (T' \leq Time < T' + Len')) \ \mid \\
\qquad \mathsf{inform\_user}(\ \texttt{failed to schedule} :, Ident, Time, Len, Loc, Att)
\end{array}
$$

2.7 Intelligent Agents

Intelligent agents in 3APL are entities which represent their environment by means of their beliefs, control this environment by means of actions which are part of their goals or plans, and manipulate their goals using practical reasoning rules. Goals keep the representation of the environment up to date by performing belief updates. The dynamic components of an agent are its beliefs and goals. During the operation of an agent these are the only components which can change. The set of practical reasoning rules associated with an agent does not change during the operation of an agent. The expertise of an agent, defined by the set of basic actions it is able to perform, also remains fixed during the lifetime of an agent. The dynamic part of an agent program is called the *mental state* of the agent. The static part of an agent deals with the specification of action and planning capabilities.

A mental state consists of two components. The first component is called the *goal base* and consists of the set of adopted goals or plans of the agent. The second component is called the *belief base* of the agent and consists of the agent's current beliefs.

Definition 2.7.1 *(mental state)*
A *mental state* is a pair $\langle \Pi, \sigma \rangle$, where

- $\Pi \subseteq \mathsf{Goal}$ is a *goal base*, i.e. a set of goals, and
- $\sigma \subseteq \mathcal{L}$ is a *belief base*.

Note that we do not allow any goal variables in the mental state of an agent, but only allow goals from Goal which do not contain occurrences of goal variables. The condition on the body of rules in definition 2.6.2 guarantees goal variables are never introduced into the goal base.

Convention 2.7.2 We use Π to denote a goal base, and σ to denote a belief base. We use Γ to denote a set of practical reasoning rules, also called a *PR-base*.

To program an agent means to specify its initial mental state, including its initial beliefs and goals, and to write a set of practical reasoning rules which define the know-how of the agent. Thirdly, it is also important to specify the basic actions which define the expertise of the agent. An agent may have more than one goal at any time. Multiple goals are executed in parallel. A 3APL agent thus is a multi-threaded entity. Finally, an agent has a *name* to identify the agent.

Definition 2.7.3 *(intelligent agent)*
An *intelligent agent* is a quadruple $\langle a, \Pi_0, \sigma_0, \Gamma \rangle$ where

- a is the *name* of the agent,
- Π_0 is the *initial goal base*,
- σ_0 is the *initial belief base*, and
- Γ is a *PR-base*.

Example 2.7.4 As an example agent, consider a personal digital assistant for managing an agenda. Such an agent program would include the components discussed in the previous examples.

- the goal base: $\{maintain\_agenda\}$, where $maintain\_agenda$ is a top level goal to maintain the agenda of the user,
- the belief base contains information about travelling and (a number of) integrity constraints discussed above:
  $\{transport(train, utrecht, amsterdam, 0:45), \ldots\}$,
- the PR-base contains a number of PR-rules as illustrated in the previous examples. These rules should be extended with rules to implement the top level goal $maintain\_agenda$.

2.8 Summary

3APL provides a general framework for programming situated agents. In the agent programming language 3APL three conceptual levels are distinguished: beliefs, goals, and practical reasoning rules. At the most basic level, the beliefs of an agent represent the current situation from the agent's point of view. At the second level, the execution of goals operate on the belief base of an agent by adding and deleting information. At the third level, practical reasoning rules supply the agent with reflective capabilities to modify its goals. This cleanly separates the different types of updating. From a more traditional perspective, the beliefs of the agent correspond to the state of the system and the goal base of an agent represents the program which is being executed. Because of the reflective capabilities of agents to modify their goals in arbitrary ways by means of rules, however, agents are not just programs in the traditional sense, but are *self-modifying* programs. This is a distinguishing feature of intelligent agents in the agent language 3APL.

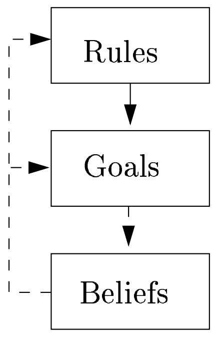

Figure 2.1: Levels of Symbolic Operators

The three levels of symbolic operators incorporated in 3APL are illustrated in figure 2.1. The dotted arrows indicate the input from the belief and goal base that is required to be able to apply operators at higher levels.

A basic architecture is associated with the language as illustrated in figure 2.2. In this basic architecture, different technologies that provide the reasoning capabilities, suitable for example for dealing with uncertainty, or for diagnosing, can be integrated, in line with our philosophy that these features are plug-in features of 3APL agents.

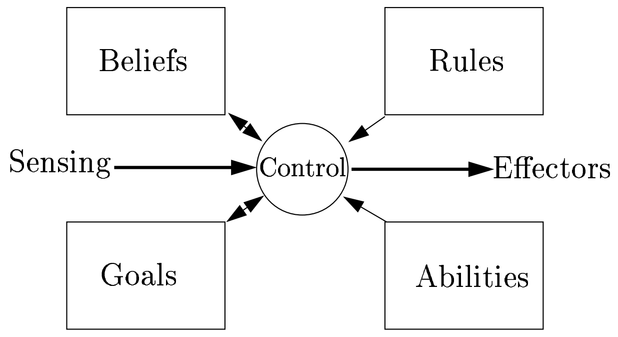

Figure 2.2: Agent Architecture

The architecture of the system is based on a modular design to make such integration as easy as possible. The programming language 3APL itself specifies the control, the execution of actions, and the application of rules to goals. In figure 2.2, we have also incorporated input from (sensing) and output to (effectors) the environment. In a multi-agent setting, these inputs and outputs may also be thought of as communication channels.

Summarising, 3APL is a combination of imperative and logic programming. Whereas imperative programming constructs are used to program the usual flow of control from imperative programming and update the current beliefs of the agent by executing basic actions, logic programming implements the querying of the belief base of the agent and the parameter mechanism of the language based on computing bindings for variables.

CHAPTER 3

Operational Semantics

The dynamics of an agent corresponds to changes in the mental state of that agent. In this chapter, we define the *operational semantics* of 3APL which formalises the dynamics of agents. The semantics specifies how the operation of an agent affects the mental state of that agent. The semantics we use here is a *transition semantics* defined by means of so-called transition systems. Operational semantics provides a constructive approach to semantics, in contrast with a denotational semantics which provides a more abstract, mathematical type of semantics (Tennent 1991).

3.1 Labelled Transition Systems

Transition systems are a means to define the *operational semantics* of a programming language (Plotkin 1981). A *transition system* consists of a set of derivation rules for deriving transitions that are associated with an agent. A transition corresponds to a *single computation step*. Such derivation rules are also called *transition rules*. A set of transition rules can be viewed as an inductive definition of a *transition relation* $\longrightarrow$, similar to a definition of a logical theory by means of logical axioms and derivation rules. The relation $\longrightarrow$ defined by a transition system is the smallest relation which contains all the axioms of the system and all conclusions which are derivable by using these axioms and transition rules. A transition relation is a relation on so-called *configurations*. We will encounter different types of configurations that correspond to different levels of abstraction. A *multi-agent level*, a *single agent level*, and a plan or *goal level* will be distinguished.

The general format of a transition rule consists of a (possibly empty) set of premises and a conclusion derivable from these premises. Both premises and conclusion are transitions. In case the set of premises is empty, the transition rule is also called an *axiom* of the transition system. Auxiliary conditions may be used to select a particular subset of transitions that may be used in a transition rule as premises or conclusion. A transition rule then looks like this:

$$\frac{C_1 \longrightarrow C_1', \ldots, C_n \longrightarrow C_n'}{C_{concl} \longrightarrow C_{concl}'} \; \textit{conditions}$$

Most of the time the conditions on the type of transitions allowed as premises or conclusion are listed above the line.

The set of transition rules associated with each programming construct of a programming language is a specification of the meaning of that construct. For example, a transition rule for sequential composition ; that specifies that π_1 in $\pi_1; \pi_2$ must be executed first, defines the (operational) meaning of the sequential composition. A transition semantics specifies what type of computation steps a program can perform and indirectly specifies the operations that are required to execute a program. In fact, a transition semantics can be viewed as specifying an *abstract machine* on which agent programs can be executed (Nielson & Nielson 1992).

To distinguish between different *types of transitions*, we also associate *labels* with transitions. A labelled transition looks like

$$C \xrightarrow{l} C'$$

These distinctions of transitions by means of labels serve several purposes. For example, labels allow an elegant modelling of communication, and in chapter 4 labels will be used to define priorities on computation steps. They will also be used to establish a number of simulation results with other programming languages in part II. Besides labels, also other types of information like, for example, bindings for variables may be associated with transitions for bookkeeping purposes.

3.2 Execution at the Multi-Agent Level

We define three transition relations corresponding to three levels of execution. The first transition relation defines what it means to execute a multi-agent system. Multi-agent execution is called *execution at the multi-agent level.* Multi-agent execution is defined in terms of single agent execution which is defined by a second transition relation. Agent execution is called *execution at the agent level.* This second transition relation, in turn, is defined in terms of a third transition relation that defines what it means to execute a single goal or plan. Goal execution is called *execution at the goal level.* Although we define three transition relations, we will use the same symbol $\longrightarrow$ to denote each of these relations. The three transition relations are relations on different types of configurations.

In the multi-agent case, a configuration is a set of agents. A multi-agent system thus is identified with a set of agents. In a multi-agent system, we assume that each agent has a unique identity (a name) which distinguishes it from the other agents. Of course, agents may also differ with respect to other features. They may have different expertises, use different knowledge representation languages, and have different responsibilities, tasks and plan libraries.

Definition 3.2.1 *(multi-agent system)*
A *multi-agent system* is a finite set $\mathcal{M} = \{A_1, \ldots, A_n\}$ of intelligent agents $A_1 = \langle a_1, \Pi_1, \sigma_1, \Gamma_1 \rangle, \ldots, A_n = \langle a_n, \Pi_n, \sigma_n, \Gamma_n \rangle$ such that for each pair of agents $A_i, A_j, i \neq j$ we have that $a_i \neq a_j$, that is, agents have different names.

A multi-agent system is executed by executing the agents of the system in parallel. Semantically, parallelism is modelled by an *interleaving semantics.* An interleaving semantics for parallelism interleaves the computation steps of the agents. Every computation step at the multi-agent level derives from a single computation step at the agent level. One of the agents in the multi-agent system is selected for execution, that agent is transformed by executing it, and accordingly, the multi-agent system is updated by replacing the selected agent by the resulting agent. At the multi-agent level, we do not associate labels with transitions. We will not need to distinguish transitions at this level.

Definition 3.2.2 *(transition rule for multi-agent systems)*
Let $\{A_1, A_2, \ldots, A_n\}$ be a multi-agent system.

$$\frac{A_i \xrightarrow{\tau} A_i' \text{ for some } i, 1 \leq i \leq n}{\{A_1, \ldots, A_i, \ldots, A_n\} \longrightarrow \{A_1, \ldots, A_i', \ldots, A_n\}}$$

The label τ associated with the transition at the agent level classifies the transition as a transition *internal* to the agent. The fact that the transition is internal to the agent means that the computation step that is involved has no effects on the states of the other agents. Of course, if an agent would only perform such τ-transitions it would be acting as if there were no other agents. In chapter 5, we will extend the basic language with *communication primitives* which allow the agents in a multi-agent system to interact.

3.3 Parameter Mechanism, Variables and Substitutions

Before we can define execution at the agent and goal levels, we need to define a number of notions related to the *parameter mechanism* in 3APL. Because we assume that our agents are equipped with a logical basis - a language for knowledge representation with an associated inference mechanism - as in logic programming, we need to define the notion of a *substitution.* Substitutions play an important role in the operational semantics of the language and are used to formally define the parameter mechanism. Since 3APL has two types of variables, two types of substitutions are defined. We use $\theta, \theta', \ldots, \gamma, \eta$ to denote substitutions.

Definition 3.3.1 *(substitution for first order variables)*

- A *substitution θ for first order variables* is a *finite set of pairs* (also called *bindings*) of the form $x_i = t_i$, where $t_i \in \mathsf{Term}$ is a term bound to variable $x_i \in \mathsf{Var}$, and $x_i \neq x_j$ for every $i \neq j$, and $x_i \notin \mathsf{Free}(t_j)$, for any i and j,

- A *ground* substitution θ is a substitution such that for every pair $x = t \in \theta$ the term t is ground, i.e. $\mathsf{Free}(t) = \varnothing$,
- The domain of θ, denoted by $dom(\theta)$, is the set of variables x for which θ contains a pair $x = t$.

From the definition of a substitution, we can derive the notion of an *answer*: a substitution θ is an answer for φ relative to a belief base $\sigma \subseteq \mathcal{L}$ iff $\sigma \models \varphi\theta$ and $dom(\theta) = \mathsf{Free}(\varphi)$.

Definition 3.3.2 *(substitution for goal variables)*
A *substitution* η *for goal variables* is a *finite set of pairs* of the form X/π where $X \in \mathsf{Gvar}$ and $\pi \in \mathsf{Goal}$. The domain of η, denoted by $dom(\eta)$, is the set of variables X such that η contains a pair X/π.

A *substitution* then is a set consisting of bindings for first order variables as well as goal variables as defined in definitions 3.3.1 and 3.3.2. A substitution thus is the union of a substitution for first order variables and goal variables. The application of a substitution to a syntactic expression is defined informally as the simultaneous replacement of expressions bound to a variable for that variable. A formal definition of the application of substitutions to formulas can be found in (Lloyd 1987)).

Definition 3.3.3 *(application of substitution)*
Let e be any syntactic expression, and θ be a substitution. Then $e\theta$ denotes the expression where *all* free variables x in e for which $x = t \in \theta$ or variables $X \in \mathsf{Gvar}$ for which $X/t \in \theta$ are *simultaneously* replaced with t.

3.4 Execution at the Agent Level

A configuration at the agent level simply is an agent and consists of the four components of an agent. The first component is the name of the agent. The next two components correspond to the current mental state of an agent, its goal base and its belief base. The fourth component consists of the rule base of the agent. Because the set of practical reasoning rules in the rule base remains fixed during computation, however, we do not explicitly list this part of a configuration nor do we mention the agent's name anymore below.

The agent level in the language 3APL consists of the mental state of the agent, that is, its goal and belief base. At this level, an agent may execute an action by choosing a plan from its goal base and selecting an action to execute from this plan. An agent may pick any plan in its current goal base that is enabled (can be executed), execute it, and update its mental state accordingly. Alternatively, an agent may select a practical reasoning rule which is applicable to a goal in its goal base and apply the rule to the goal. The latter mechanism supplies the agent with reflective capabilities concerning its goals.

Computation steps at the agent level are derived from computation steps at the goal level. Since computation steps at the goal level are computation steps of a single goal, a configuration at the goal level is a pair $\langle \pi, \sigma \rangle$ where π is a goal and σ is a belief base. Again we suppress the rule base from the configuration. At the agent level, the set of goals in the goal base of an agent are executed in parallel. The parallel execution of multiple goals is, as before in the case of multi-agent systems, modelled by the interleaving of the computation steps of the different goals.[1] Because an agent executes multiple goals in parallel, an agent is a multi-threaded system.

Convention 3.4.1 We use $V \subseteq \mathsf{Var}$ to denote an arbitrary set of first order variables.

Definition 3.4.2 *(transition rule for single agents)*
Let $\Pi = \{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots\} \subseteq \mathsf{Goal}$, θ be a ground substitution, and $V = \mathsf{Free}(\Pi)$. Then:

$$\frac{\langle \pi_i, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_i', \sigma' \rangle}{\langle \{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots\}, \sigma \rangle \xrightarrow{l} \langle \{\pi_0, \ldots \pi_{i-1}, \pi_i', \pi_{i+1}, \ldots\}, \sigma' \rangle}$$

Note the similarity of the transition rule for single agents and the transition rule which defines computation steps of multi-agent systems. An agent is executed by selecting one of its goals and executing the selected goal. The executed goal consecutively is replaced with the resulting new goal and any changes to the belief base due to the execution of the goal are passed on to the agent level.

There is no communication through shared variables at the agent level. Occurrences of the same free first order variable in two different goals thus are unrelated. Communication between goals proceeds by means of the shared belief base. Because goals change and inspect the same belief base, the belief base of an agent from the point of view of a goal is similar to a blackboard. The beliefs of an agent thus can also be used to communicate information between goals.

Finally, the purpose of the set V of first order variables in the premise of the transition rule is explained as follows. Because the application of practical reasoning rules may introduce new occurrences of first order variables into the goal base, as we will see below, we have to take care that no new (implicit) bindings are created between variables. For this reason, the set of all free variables in the goal base is a parameter of the transition in the premise, which is used to prevent the introduction of any new implicit bindings in the transition rule for the application of practical reasoning rules.

[1] An interleaving semantics for the parallel execution of goals provides a useful abstraction for modelling parallel execution of goals in practice. An interleaving semantics requires that an implementation of parallelism in the language by true concurrent execution of goals also provides a strategy for ruling out any possible conflicts which might occur if two or more goals simultaneously try to access the same data structure (update a particular belief or variable, in our case). However, an interleaving semantics abstracts from such implementation issues.

3.5 Execution at the Goal Level

In this and the next section, we define execution at the goal level. The transition rules below define possible computation steps for a single goal given a current belief base. In this section, transition rules that specify the semantics of basic actions, test goals, sequential composition, nondeterministic choice and parallel composition are provided. In the next section, transition rules for the application of practical reasoning rules are discussed.

Convention 3.5.1 We use E to denote *successful termination*, and identify $E;\ \pi$, $E\|\pi$ and $\pi\|E$ with π. Moreover, $\Pi \cup \{E\}$ is identified with Π. The symbol E is used to denote the end of execution. Alternatively, E can be thought of as the 'empty goal'.

The basic action repertoire of an agent defines the skills or expertise of that agent. As discussed in the previous chapter, basic actions are updates on the belief base of an agent. The semantics for these actions is a parameter of the transition semantics. We did not specify a fixed set of actions as part of the language 3APL, but allow the programmer to select and define any set of actions which are most suitable to him. The semantics of basic actions therefore is assumed to be given and a *transition function* is used to abstract from any specific set of actions. A transition function for basic actions is a mapping from a basic action and a belief base to a new belief base and defines the type of update a basic action is.

Definition 3.5.2 *(semantics of basic actions)*
A *transition function* $\mathcal{T}$ is a (partial) function of type : $\mathsf{Bact} \times \mathcal{L} \to \mathcal{L}$.

Example 3.5.3 The semantics of the action

$$\mathsf{ins}(meet(Ident, Time, Len, Loc, Att))$$

for a belief base σ (with a closed world assumption associated with *meet*) of the form

$$\begin{aligned}
&\forall\, Id, T, Len, Loc, A(meet(Id, T, Len, Loc, A) \leftrightarrow \\
&((Id = i1 \wedge T = t1 \wedge Len = d1 \wedge Loc = l1 \wedge A = a1) \vee \ldots \vee \\
&(Id = iN \wedge T = tN \wedge Len = dN \wedge Loc = lN \wedge A = aN))) \wedge \ldots
\end{aligned}$$

can be defined by $\mathcal{T}(\mathsf{ins}(meet(Ident, Time, Len, Loc, Att)), \sigma) = \sigma'$ where

$$\begin{aligned}
&\sigma' = \\
&\forall\, Id, T, Len', Loc', A(meet(Id, T, Len', Loc', A) \leftrightarrow \\
&((Id = i1 \wedge T = t1 \wedge Len' = d1 \wedge Loc' = l1 \wedge A = a1) \vee \ldots \vee \\
&(Id = aN \wedge T = tN \wedge Len' = dN \wedge Loc' = lN \wedge A = aN) \vee \\
&(Id = Ident \wedge T = Time \wedge Len' = Len \wedge Loc' = Loc \wedge A = Att))) \wedge \ldots
\end{aligned}$$

for all ground instances of the variables $Ident, Time, Len, Loc, Att$ such that σ' is consistent; for all belief bases σ'' equivalent to σ, define

$$\mathcal{T}(\mathsf{ins}(meet(Ident, Time, Len, Loc, Att)), \sigma'') = \sigma'$$

; otherwise, $\mathcal{T}$ is undefined. Informally, this definition specifies that the ins action expands the belief base with a new belief concerning the agenda if this is consistent and otherwise the action is not enabled.

A number of constraints are imposed on transition functions. First of all, the update performed by a basic action on logically equivalent belief bases should result in logically equivalent belief bases again. That is, if σ and σ' are logically equivalent, then $\mathcal{T}(\mathsf{a}, \sigma)$ is defined if $\mathcal{T}(\mathsf{a}, \sigma')$ is defined and vice versa. Moreover, if $\mathcal{T}(\mathsf{a}, \sigma)$ is defined it must be logically equivalent with $\mathcal{T}(\mathsf{a}, \sigma')$. Secondly, we assume that a transition function is only defined for *ground* basic actions, that is, for basic actions without occurrences of free variables. The reason for this is that it is not clear what the meaning of non-ground basic actions should be. For example, what does it mean to execute the action $\mathsf{ins}(\mathit{meet}(\mathit{Ident}, \mathit{Time}, 1:00, \mathit{utrecht}, \mathit{Att}))$?

And third, we do not allow updates on functions. For example, if $f(a) = b$ is a definition of a function f on argument a in the belief base, then no basic action is allowed to change this definition, into, for example, $f(a) = c$ where $c \neq b$. If it would be allowed to update functions, we could have the following: Suppose, initially, $f(a) = 1$ and $p(1) \wedge \neg p(2)$ holds, and that action a is allowed to update $f(a)$ such that $f(a) = 2$. In that case, the goal $x = f(a)?;\ \mathsf{a};\ p(x)?$ would only successfully terminate if x is bound to 1 and not to $f(a)$. If function updating is allowed, it thus makes a difference in what way is referred to a particular value, for example, by 1 or $f(a)$. For simplicity, here we prefer a parameter mechanism with explicit value passing over a mechanism which would only pass textual references.

Given a transition function $\mathcal{T}$ which specifies the update performed by a basic action, we can now define the semantics of basic actions. The execution of a basic action $\mathsf{a}(\vec{t})$ amounts to updating the beliefs in accordance with the transition function. The execution of a basic action does not compute any bindings for variables which explains the fact that the empty substitution is associated with a transition due to the execution of a basic action.

Definition 3.5.4 *(transition rule for basic actions)*

$$\frac{\mathcal{T}(\mathsf{a}(\vec{t}), \sigma) = \sigma'}{\langle \mathsf{a}(\vec{t}), \sigma \rangle_V \xrightarrow{\tau}_{\varnothing} \langle E, \sigma' \rangle}$$

A test goal $\varphi?$ computes bindings for the free variables in the condition φ that is being tested. It is *enabled*, i.e. it can be executed, only if some instance of the condition φ is entailed by the current beliefs. If no such instance can be found, nothing happens and the test is blocked. The bindings that are computed are recorded in a substitution θ. This substitution is associated with the transition for later reference and then can be used to pass computed bindings on from one subgoal to another subgoal. A test goal $\varphi?$ *initialises* the free variables in the condition φ and binds a term to a variable, but once a term is bound to a variable this binding cannot be changed anymore like in imperative programming. The bindings $x = t$ that are computed are required to be ground, i.e. t is required to be ground. A test is evaluated relative to the current belief base. From a logic programming perspective, the belief base can be viewed as a logic program which is used to compute the bindings. A test does not change the beliefs of the agent.

Definition 3.5.5 *(transition rule for tests)*
Let θ be a ground substitution such that $dom(\theta) = \mathsf{Free}(\varphi)$.

$$\frac{\sigma \models \varphi\theta}{\langle \varphi?, \sigma\rangle_V \xrightarrow{\tau}_\theta \langle E, \sigma\rangle}$$

Example 3.5.6 In case the current belief base of an agent contains the proposition

$$meet(meeting, 10:00, 1:00, utrecht, \{peter, john, user\})$$

then the test

$$meet(meeting, Time, 1:00, utrecht, Att)?$$

can return the binding $Time = 10:00$. It will return this binding if there is only one meeting of 1:00 in Utrecht stored in the agenda. If more than one meeting takes place at the same time and place, also alternative bindings can be returned. More formally, there may be different answers $\theta \neq \theta'$ such that both $\sigma \models \varphi\theta$ and $\sigma \models \varphi\theta'$; in that case, one of these substitutions is nondeterministically chosen.

A sequential goal is executed by executing the subgoals consecutively. A single computation step of the first subgoal is performed first. Any changes to the belief base due to this computation step are recorded. Any bindings computed in the step are passed on to the second subgoal. The resulting new goal consists of the transformed first subgoal sequentially composed with the second subgoal (to which the computed substitution has been applied). The substitution does not need to be applied to π_1' because we have that $\pi_1'\theta = \pi_1'$ below.

Definition 3.5.7 *(transition rule for sequential composition)*

$$\frac{\langle \pi_1, \sigma\rangle_V \xrightarrow{l}_\theta \langle \pi_1', \sigma'\rangle}{\langle \pi_1;\ \pi_2, \sigma\rangle_V \xrightarrow{l}_\theta \langle \pi_1';\ \pi_2\theta, \sigma'\rangle}$$

The bindings computed by a test are used in the remaining computation. We illustrate this by an example which also illustrates the transition rule for sequential composition.

Example 3.5.8 Consider the sequential goal

$$\begin{array}{l} location(user, Loc, 10:00)?; \\ meet(Id, 10:00, Len, Loc, Att) \wedge user \in Att? \end{array}$$

and assume the location in the belief base at 10:00 is Utrecht since a meeting with Peter and John is scheduled at that time. Then the first step is to execute the test $location(user, Loc, 10:00)?$ which returns a binding $Loc = utrecht$. The remaining goal consists of the test $meet(Id, 10:00, Len, utrecht, Att)?$. Evaluating this test with respect to the current belief base then results in bindings $Id = meeting$, $Len = 1:00$, and $Att = \{peter, john, user\}$.

The execution of a nondeterministic choice goal amounts to selecting one of the subgoals that is enabled, execute this goal, and drop the other subgoal. The semantics of nondeterministic choice is defined by two transition rules, one rule in which the left branch of the choice goal is chosen and another rule in which the right branch of the choice goal is selected.

Definition 3.5.9 *(transition rules for non-deterministic choice)*

$$\frac{\langle \pi_1, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_1', \sigma' \rangle}{\langle \pi_1 + \pi_2, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_1', \sigma' \rangle} \qquad \frac{\langle \pi_2, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_2', \sigma' \rangle}{\langle \pi_1 + \pi_2, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_2', \sigma' \rangle}$$

As before, the execution of a parallel goal $\pi_1 \| \pi_2$ is modelled by interleaving. In this respect, a parallel goal $\pi_1 \| \pi_2$ is similar to the execution of two single goals π_1 and π_2 in the goal base. There is, however, one important difference. The subgoals in a parallel goal $\pi_1 \| \pi_2$ *communicate through shared variables.*

Any bindings that are computed and recorded in a substitution θ during the process of executing one of the parallel subgoals are also passed on to the other subgoal. By applying the computed substitution θ to the other subgoal a communication mechanism between multiple goals is created based on the sharing of variables. This type of communication we will also call *communication at the goal level.*

Definition 3.5.10 *(transition rules for parallel composition)*

$$\frac{\langle \pi_1, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_1', \sigma' \rangle}{\langle \pi_1 \| \pi_2, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_1' \| \pi_2\theta, \sigma' \rangle} \qquad \frac{\langle \pi_2, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_2', \sigma' \rangle}{\langle \pi_1 \| \pi_2, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_1\theta \| \pi_2', \sigma' \rangle}$$

Example 3.5.11 As a simple example to illustrate the communication between parallel goals, consider the goal $p(x)? \| \mathsf{del}(p(x))$ where the subgoal $p(x)?$ communicates a value for x to the other subgoal. (Recall a basic action with free variables cannot be executed). The subgoal $p(x)?$ thus is the producing goal (it outputs the value) and the other subgoal is the receiving goal. Suppose that a is bound to x by the test $p(x)?$. Then the new goal becomes $\mathsf{del}(p(a))$ and by executing this action $p(a)$ is removed from the agent's beliefs.

3.6 Application of Practical Reasoning Rules

Practical reasoning rules operate on the goals of an agent. A practical reasoning rule $\pi_h \leftarrow \varphi \mid \pi_b$ is applicable if the head of the rule unifies with a (subgoal of a) current goal of the agent and the guard is entailed by the current beliefs. Two transition rules are required to define the application of rules due to the different nature of rules with an empty head and rules which do not have an empty head. The semantics of rules with an empty body can be viewed as a special case of the semantics of rules with nonempty bodies, where the goal which unifies with the head of such a rule is dropped and this is interpreted as successful termination. Alternatively, a rule of the form $\pi_h \leftarrow \varphi$ can be identified with $\pi_h \leftarrow \varphi \mid E$ where E is the 'empty goal'.

In the transition rule that defines the application of a practical reasoning rule, two notions play an important role: the notion of a *variant* and the notion of *matching*. Both notions apply to expressions. An expression e is a variant of another expression e' in case e can be obtained from e' by renaming of variables (for a formal definition cf. Lloyd (1987)). The matching of two expressions is not formally introduced here. For simplicity, we will assume here that two goals π_1 and π_2 *match*, denoted by $\pi_1 \precapprox \pi_2$, if they are identical. In chapter 4, a more precise and formal definition of $\precapprox$ is provided.

Definition 3.6.1 *(transition rule for rule application)*
Let η be a most general unifier for π and π_h such that $\pi \precapprox \pi_h\eta$, and θ be a ground substitution such that $dom(\theta) \subseteq \mathsf{Free}(\varphi\eta)$.

$$\frac{\sigma \models \varphi\eta\theta}{\langle \pi, \sigma \rangle_V \xrightarrow{\tau}_\theta \langle \pi_b\eta\theta, \sigma \rangle}$$

where $\pi_h \leftarrow \varphi \mid \pi_b$ is a *variant* of a PR-rule in the PR-base Γ of the agent such that no free variables in the rule occur in V.

The application of a PR-rule defined by the transition rule is explained as follows. The first step is to check whether or not the current (sub)goal π is an instance of the head π_h of the rule. If this is the case, there is a most general unifier η with bindings for *all* goal variables in π_h (because π does not contain goal variables) and possibly some bindings for first order variables. The bindings for the first order variables may represent *input values* supplied to the body of the rule but may also rename variables. The computation of the substitution η is based on pattern-matching. The second step is to check whether the guard is entailed by the current beliefs. The evaluation of a guard is analogous to the evaluation of a test, but in the case of a guard the input values which are stored in the substitution η are used first to instantiate the guard. The evaluation of the guard may compute new bindings θ for free variables in $\varphi\eta$. Finally, the original goal π is replaced by the body of the rule which has been instantiated with the bindings from $\eta\theta$. Since η can only contain bindings for first order variables which do not occur in the goal base of the agent (the rule is a variant), we do not associate these bindings with the transition; the substitution θ which is computed by evaluating the guard of the rule, however, may contain bindings for variables in the goal base since η may have renamed variables in the rule to variables that occur in the goal base. Therefore, θ is associated with the computation step.

We remark here that the restriction on goal variables (cf. definition 2.6.2) such that all goal variables in the body of a rule must also occur in the head of the rule implies that no goal variables are introduced into the goal base of an agent by means of rule application, because the most general unifier η binds all goal variables to goals.

The transition rule for PR-rule application only specifies *how* to apply a rule to a goal. It does not specify, however, *which* applicable rule - in case there is more than one - should be applied. It also does not determine which substitution θ is applied (in case there are $\theta \neq \theta'$ such that $\sigma \models \phi\eta\theta$ and $\sigma \models \phi\eta\theta'$).

Special care needs to be taken to avoid introducing any new implicit bindings between variables by applying a rule and replacing a goal by the body of a rule. The set V is used to prevent this. V consists of all the free first order variables of the current goal base. The introduction of new implicit bindings is precluded by the requirement that a variant of a practical reasoning rule must be used in which new variables - which do not occur in the current goal base - have been substituted for the free variables.

Example 3.6.2 We illustrate what can go wrong if variables in a rule are not renamed appropriately. Recall the rule for scheduling an activity from example 2.6.4:

$$\begin{array}{l}
\mathit{schedule}(\mathit{Ident}, \mathit{Time}, \mathit{Len}, \mathit{Loc}, \mathit{Att}) \\
\leftarrow \mathit{location}(\mathit{user}, \mathit{FromLoc}, \mathit{Time}) \wedge \mathit{user} \in \mathit{Att} \ \mid \\
\quad \mathit{transport}(\mathit{Means}, \mathit{FromLoc}, \mathit{Loc}, \mathit{DurTrans})?; \\
\quad \mathsf{ins}(\mathit{meet}(\mathit{Means}, \mathit{Time} - \mathit{DurTrans}, \mathit{DurTrans}, \mathit{FromLoc}, \mathit{user})); \\
\quad \mathsf{ins}(\mathit{meet}(\mathit{Ident}, \mathit{Time}, \mathit{Len}, \mathit{Loc}, \mathit{Att}))
\end{array}$$

Now suppose one of the current goals of the agent is:

$$\begin{array}{l}
\mathit{schedule}(\mathit{IA\_course}, 11:00, 1:00, \mathit{utrecht}, \mathit{students}); \\
\mathit{transport}(\mathit{Means}, \mathit{utrecht}, \mathit{amsterdam}, \mathit{DurTrans})?; \\
\ldots
\end{array}$$

and among other things the agent believes: $\mathit{location}(\mathit{user}, \mathit{utrecht}, 11:00) \wedge \mathit{user} \in \mathit{students}$.

In that case, if we do not rename the variables in the plan rule for scheduling a meeting, then by applying this rule to the goal we would end up with the following new plan, in which the variable *Fromloc* has been instantiated with the computed binding $\mathit{Fromloc} = \mathit{utrecht}$:

$$\begin{array}{l}
\mathit{transport}(\mathit{Means}, \mathit{utrecht}, \mathit{utrecht}, \mathit{DurTrans})?; \\
\mathsf{ins}(\mathit{meet}(\mathit{Means}, 11:00 - \mathit{DurTrans}, \mathit{DurTrans}, \mathit{utrecht}, \mathit{user})); \\
\mathsf{ins}(\mathit{meet}(\mathit{IA\_course}, 11:00, 1:00, \mathit{utrecht}, \mathit{students})); \\
\mathit{transport}(\mathit{Means}, \mathit{utrecht}, \mathit{amsterdam}, \mathit{DurTrans})?; \\
\ldots
\end{array}$$

In the resulting plan, an implicit binding is introduced between the first and second occurrence of the variable *Means* and the first and second occurrence of the variable *DurTrans*. However, the type of transportation required to get from Utrecht to Utrecht (none) and the time it takes (none) are obviously different from the type of transportation required to get from Utrecht to Amsterdam (train, bus, etc.) and the time it takes. No link between these different entities was intended, and would not have been introduced if the variables in the rule would have been renamed appropriately.

Example 3.6.3 We illustrate the application of the practical reasoning rule from example 2.6.6 to deal with a failure of the scheduling of an activity. For ease of exposition, we repeat the practical reasoning rule to deal with a failure to schedule the time needed for travelling first.

$$
\begin{array}{l}
X;\ \mathsf{ins}(\mathit{meet}(\mathit{Ident}, \mathit{Time}, \mathit{Len}, \mathit{Loc}, \mathit{Att})) \\
\leftarrow \mathit{meet}(\mathit{Id}', T', \mathit{Len}', \mathit{Loc}', \mathit{Att}')\wedge \\
\quad ((\mathit{Time} \leq T' < \mathit{Time} + \mathit{Len}) \vee (T' \leq \mathit{Time} < T' + \mathit{Len}')) \mid \\
\qquad \mathsf{inform\_user}(\texttt{failed to schedule :}, \mathit{Ident}, \mathit{Time}, \mathit{Len}, \mathit{Loc}, \mathit{Att})
\end{array}
$$

To illustrate the application of this failure rule, suppose that the agent has applied the plan rule for scheduling a meeting to the scheduling goal of example 2.6.4, already has executed the test in that plan, and ended up with the remaining goal:

$$
\begin{array}{l}
\mathsf{ins}(\mathit{meet}(\mathit{train}, 10:15, 0:45, \mathit{amsterdam}, \mathit{user})); \\
\mathsf{ins}(\mathit{meet}(\mathit{IA\_course}, 11:00, 1:00, \mathit{utrecht}, \mathit{students}))
\end{array}
$$

To create a situation where a conflict arises if the course would be scheduled, assuming that the user is a member of *students*, also suppose that the agent believes that:

$$\mathit{meet}(\mathit{meeting}, 11:30, 1:30, \mathit{utrecht}, \{\mathit{peter}, \mathit{user}\})$$

It is clear that the agent cannot schedule the course into the agenda because of the meeting that already has been scheduled at 11:30. To see how the failure rule deals with this, first note that the head of the failure rule can be unified with the (remaining part of the) scheduling plan. The most general unifier η is $\{\mathit{Ident} = \mathit{IA\_course}, \mathit{Time} = 11:00, \mathit{Len} = 1:00, \mathit{Loc} = \mathit{utrecht}, \mathit{Att} = \mathit{students}, X/\mathsf{ins}(\mathit{meet}(\mathit{train}, 10:15, 0:45, \mathit{amsterdam}, \mathit{user}))\}$. Note that this unifier also contains a binding for the goal variable X. The second step is to check whether an instance of the guard is entailed by the agent's beliefs. It is not difficult to see that the agent's beliefs entail

$$
\begin{array}{l}
\mathit{meet}(\mathit{meeting}, 11:30, 1:30, \mathit{utrecht}, \{\mathit{peter}, \mathit{user}\})\wedge \\
((11:00 \leq 11:30 < 11:00 + 1:00)\vee \\
\quad (11:30 \leq 11:00 < 11:30 + 1:30))
\end{array}
$$

which is an instance of the guard (note that variable *Time* and *Len* are bound by the unifier η). The guard thus returns a substitution θ which binds the variables $Id', T', Len', Loc', Att'$ in the guard:

$$\theta = \{Id' = meeting, T' = 11:30, Len' = 1:30, Loc' = utrecht, \\ Att' = \{peter, user\}\}$$

Therefore, the failure rule is applicable and in the third and last step the original goal consisting of the two ins actions is replaced by the body of the rule, where free variables are instantiated by $\eta\theta$. The resulting goal is:

$$\mathsf{inform\_user}(\texttt{failed to schedule}:, IA\_course, 11:00, 1:00, \\ utrecht, students)$$

which informs the user that the agent was unable to schedule the course in the agenda. (Note that in this example it is unnecessary to rename any variables in the failure rule.)

The last transition rule specifies the semantics of rules with an empty head. Most of the details are analogous to that of the transition rule for rules with a non-empty head. The main difference is that there is no unification with a current (sub)goal of the agent involved. Condition-action rules *add* new goals to the current goal base. Because the creation of a new goal changes the goal base of an agent, the semantics of condition-action rules needs to be specified at the agent level and not at the goal level.

Definition 3.6.4 *(transition rule for the application of condition-action rules)* Let Π be a goal base and θ be a ground substitution such that $dom(\theta) = \mathsf{Free}(\varphi)$.

$$\frac{\sigma \models \varphi\theta}{\langle \Pi, \sigma \rangle \longrightarrow \langle \Pi \cup \{\pi\theta\}, \sigma \rangle}$$

where $\leftarrow \varphi \mid \pi_b \in \Gamma$ is a condition-action rule in the PR-base of the agent.

No variant of a condition-action rule is used in the application of such a rule. There is no need to do so because there is no communication between free first order variables in different goals, and no interference between such variables can occur. The fact that there is no need to use a variant also has the advantage that multiple applications of a condition-action rule do not introduce multiple copies of the 'same' goal into the goal base due to the *set interpretation* of the goal base.

3.7 Conclusion

In this chapter, a detailed discussion of the operational semantics of the agent language 3APL has been presented. The formalism of transition systems was introduced, including the concept of labelling. Transitions correspond to computation steps of agents. In a labelled transition system, labels can be associated with transitions. The operational semantics of 3APL consists of three different levels. At the multi-agent level, the execution of multiple agents is formalised. At the agent level, the execution of multiple goals in the goal base of a single agent is specified. Finally, at the goal level the execution of individual goals is defined.

CHAPTER 4

Practical Reasoning Rules

The aim of this chapter is to introduce a number of techniques and tools for programming with practical reasoning rules. We study the semantics and the use of practical reasoning rules in more detail. The semantics of practical reasoning rules depends on a notion of *matching* that is made precise by means of so-called *goal trees*, which are a different representation for goals. It is useful to distinguish practical reasoning rules according to their purpose, and we propose an intuitive classification to this end. Different priorities are associated with the different classes. It is also shown how to integrate the concept of priority into the operational semantics. Several illustrations of the use of rules are provided. It is shown that practical reasoning rules can be used for creating new goals, and a technique for programming with condition-action rules is introduced. The use of rules, in particular for dealing with failure, monitoring and recovery, is investigated. A technique called postfix-labelling is introduced that facilitates the programming with practical reasoning rules with goal variables. Finally, we show how parallelism can be simulated by practical reasoning rules.

4.1 A Classification of Practical Reasoning Rules

Practical reasoning rules in 3APL serve a number of purposes. They can be used by the agent to find a plan to achieve one of its (achievement) goals. But these rules also allow an agent to *revise* its goals in almost any way it wants. The use of goal variables in practical reasoning rules provides an agent with *reflective capabilities* concerning its goals and adopted plans.

It is useful to classify different types of rules into different classes according to their purpose. Such a classification illustrates some of the ideas that motivated the introduction of these rules and may provide a guide for their use. Moreover, the relation between different classes of rules can be studied and we may, for example, assign different priorities to each class. The classification that we propose is based on common sense considerations. Therefore, we believe, it both supports and extends the application of the metaphor of intelligent agents for designing agents. The classification also illustrates the *expressive power* of practical reasoning rules (what can be done with rules).

Our classification of practical reasoning rules is a simple and intuitive one. It is partly based on the syntactic structure of rules and highlights the close correspondence between the syntactic structure and the use of rules. Of course, the classification is just one out of several alternatives, and is not derived directly from the agent programming language itself. Moreover, although it is quite natural to propose a classification that has a close correspondence with the syntactic structure of rules, it is not sound to conclude from this that there is a one-to-one correspondence of the form of a rule and its purpose. Nevertheless, we think that the classification is one of the more natural and useful ones in practice. In the classification, four types of rules are distinguished:

- the class $\mathcal{F}$ of *failure rules*,
- the class $\mathcal{P}$ of *plan rules*,
- the class $\mathcal{O}$ of *optimisation rules*, and
- the class $\mathcal{C}$ of *condition-action rules.*

Different purposes are associated with each class. In short, failure rules are designed for failure handling, plan rules for finding plans, optimisation rules for optimising current plans, and condition-action rules allow the agent to respond to the current situation and take advantage of any available opportunities. Based upon this classification, it is possible to impose an intuitive and logical order on practical reasoning rules. We propose the following *partial* ordering: $\mathcal{F} > \mathcal{P}$, $\mathcal{P} > \mathcal{O}$, $\mathcal{P} > \mathcal{R}$ (no other pairs are included in the relation $>$). Of course, alternative orderings are conceivable. But the ordering makes sense and can be motivated by the slogan 'safety first'. We now motivate the classification and priority ordering by discussing the different purposes associated with each class.

The highest priority is assigned to the class of *failure rules.* Failure rules can be used to revise plans that were adopted to achieve a particular goal. Revision is in order in case the means to achieve one of the agent's goals are no longer appropriate in the current situation the agent believes it is in. The syntactic form of failure rules is $\pi_h \leftarrow \varphi \mid \pi_b$ where π_h is any semi-goal (cf. definition 2.6.1) except for an achievement goal. The semi-goal π_h determines the class of plans that are up for revision and the guard φ of the failure rule determines in what situation revision is required. The high priority that is assigned to the class of failure rules stems from the fact that we first of all want our agents to behave in a safe way and failure rules exactly serve this purpose of avoiding failure, or cleaning up after failure.

Generally speaking, failure is due to one of two things: (i) lack of knowledge resources, and (ii) lack of control over the environment. Lack of knowledge resources and lack of control are features of any situated agent which must achieve its goals in a complex environment. Moreover, these features interact with each other: for example, a lack of information may be the cause of a lack of control, and vice versa, a lack of control may be the cause of a lack of information, because changes in the environment most of the time are unpredictable when there is a lack of control over the environment.

In general, there are three methods of responding to a (possible) failure: (i) the agent might *re-plan from scratch* and try to achieve the same goal in some other way, (ii) the agent might *repair* an adopted plan to accomplish a goal and execute this modified plan, and (iii) the agent might *drop* (part of) its *goals*. The type of response is embodied in the body π_b of the failure rule. For example, a failure due to the fact that a precondition of an adopted plan does not hold might be dealt with by interrupting the plan and inserting a plan for establishing the missing preconditions. Thus, a repair plan may be a suitable approach for missing preconditions and in that case the body of a failure rule might look like $\pi_b = \pi_{repair}; \pi_h$ where π_{repair} is the repair plan and π_h the original goal.

After dealing with behaviour that might lead to failure, an agent should use its time to find the appropriate means to achieve its achievement goals. The *class of plan rules* exactly serves this purpose. The syntactic form of plan rules is $p(\vec{t}) \leftarrow \varphi \mid \pi$. Plan rules provide a plan in the body of the rule to achieve the achievement goal in the head of the rule. The set of such rules in the rule base of an agent constitutes a plan library that the agent can consult to find a plan for an achievement goal. The priority that is assigned to plan rules stems from our concern that safety should be guaranteed first, and that only after this has been accomplished the agent should spent time deliberating about the means how to achieve its achievement goals. Nevertheless, planning may be the most natural and in any case a necessary mode of functioning of an agent.

The lowest priority is assigned to the *class of optimisation* and the *class of condition-action rules*. The syntactic form of optimisation rules is $\pi_h \leftarrow \varphi \mid \pi_b$ where π_h is a semi-goal except for an achievement goal. The form of optimisation rules thus is the same as that of failure rules. On the basis of their purpose, dealing with failure or with optimisation, respectively, it seems not possible to derive different syntactic forms for the two types of rules. The purpose of optimisation rules is to improve the performance of an agent. An optimisation rule is useful in case the *means* (represented by the head of the rule) to achieve a goal are believed to be suboptimal (indicated by the guard of the rule) and a more optimal plan (the body of the rule) to achieve the same goal is present. Presumably, without these types of rules nothing would go wrong, but more costs would be induced by the agent's current plan than necessary. If the agent has enough time to reconsider adopted plans, optimisation rules can be used to optimise the agent's actions.

The class of *condition-action rules* consists of rules with an empty head. Its syntactic form thus is $\leftarrow \varphi \mid \pi$. The application of condition-action rules does not depend on the current goals of an agent but only on the current beliefs of an agent. The guard of the rule determines in what situation the plan π is appropriate. Condition-action rules thus allow an agent to respond to particular situations. For example, a condition-action rule can be used by a personal assistant that manages the agenda of its user to inform the user of an opportunity to go to a highly recommended concert. Condition-action rules thus can be used as a means to exploit opportunities in the environment of the agent, but in this view should not use any resources (time, etc.) which are needed to deal with either failure or planning.

The classification can be used to design a particular *agent architecture* that specifies when an agent should act and when it should deliberate upon its goals. Some implications for such an architecture can be derived from our classification. Because the application of failure rules is time-critical since they serve to avoid or recover from failure, we argue here that it is necessary to apply *all* the applicable rules of this class in one cycle of an interpreter. In contrast, there is no need to apply as many as possible plan or optimisation rules. For planning and optimisation, a *least commitment strategy* (cf. below) seems more appropriate and a single rule can be selected from the latter classes in case no failure rules are applicable. The application of condition-action rules depends very much on the conditions and the opportunities associated with these conditions. Different degrees of urgency might be associated with such rules individually.

4.2 Priorities and Labelled Transitions

One way of implementing priorities is to build an architecture on top of the agent language which schedules the application of rules and the execution of actions in such a way that the priorities are accounted for. Another possibility to impose an order on the computation steps that are performed is to directly integrate such priorities into the semantics. This possibility will be explored here.

A simple and elegant way to integrate priorities into the semantics is to associate priorities with the labels associated with transitions. The idea is to prefer transitions with a specific label over transitions that are assigned other labels. For example, we can associate different labels with the application of practical reasoning rules of different classes. To do that, we slightly have to modify the transition rules for the application of rules that were introduced previously. To implement the idea we associate with the application of failure rules a label f, with plan rules a p, with that of condition-action rules a c, and with optimisation rules an o. The remaining transitions are labelled with τ as before.

The priorities of the previous section now can be associated with the labels. Technically, priorities on labels need to be taken into account at the agent level. At this level, computation steps with the highest associated level should be selected and performed first. The agent level transition rule then should be modified to incorporate this preference. This can be done by requiring that the priorities are obeyed in the premise of the transition rule.

Definition 4.2.1 *(agent level transition rule with priorities)*
Let $\Pi = \{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots\} \subseteq \mathsf{Goal}$, θ, θ' be ground substitutions, $V =$

$\mathsf{Free}(\Pi)$, and $>$ be an ordering on labels. Then:

$$\frac{\langle \pi_i, \sigma \rangle_V \xrightarrow{l}_\theta \langle \pi_i', \sigma' \rangle, \text{ and } \nexists\, l' > l, j, \sigma'' \text{ such that } \langle \pi_j, \sigma \rangle_V \xrightarrow{l'}_{\theta'} \langle \pi_j', \sigma'' \rangle}{\langle \{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots\}, \sigma \rangle \xrightarrow{l} \langle \{\pi_0, \ldots \pi_{i-1}, \pi_i', \pi_{i+1}, \ldots\}, \sigma' \rangle}$$

The transition rule directly incorporates priorities into the operational semantics. To implement the order of the previous section, we need to associate the following order with the labels: a label f has higher priority than p ($f > p$), and p has a higher than o and c ($p > o, p > c, o \not> c, c \not> o$). The τ-label associated with the execution of actions and tests, however, should also be accounted for in the order and this leaves us with a choice. A natural choice is to assign the same priority to action execution as to optimisation rules. This means that action execution, and the application of optimisation and condition-action rules have the same priority here. Throughout this chapter, we use this ordering.

4.3 Goal Trees and Matching Goals

Some of the uses of practical reasoning rules have been discussed in section 4.1, but some of the technical details of applying rules still need further clarification. In particular, the concept of *matching* used in definition 3.6.1 must be made more precise. To this end, we introduce the notion of a *goal tree* and define the concept of matching goals in terms of this notion.

Basically, a goal tree is a tree-representation of the syntactic structure of a semi-goal. A *goal tree* of a semi-goal is a labelled tree that represents the task sequence information, the choice points and the parallel subgoals in the semi-goal. The internal nodes of a goal tree are labelled with the program constructs ; , + and || to represent this information. An internal node in a goal tree that is labelled with + is called a *choice point* (of the corresponding goal). The leaves of a goal tree are labelled with basic goals and goal variables. A goal tree can be recursively generated from the syntactic structure of a semi-goal. To represent the 'empty tree', we use the 'empty goal' E.

Each goal has an associated *canonical goal tree* that corresponds to that goal. There may, however, also be other goal trees that are associated with a goal than its canonical goal tree. This is due to the fact that we want to disregard the order of subgoals in a choice or parallel goal. Consequently, in a goal tree the order of the branches from a choice point or parallel node can be switched to obtain yet another goal tree representing one and the same goal. The identification of such goal trees as representants of one and the same goal is made to guarantee the commutativity of the nondeterministic choice + and parallel composition ||.

Definition 4.3.1 *(canonical goal tree associated with π)*
Let $\pi \in \mathsf{SGoal}$ be a semi-goal. The canonical *goal tree* T_π associated with π is inductively defined as a labelled tree as follows:

(i) If π is the empty goal E, then $T_\pi = E$,

(ii) If π is a basic action $\mathsf{a}(\vec{t})$, a test $\varphi?$, an achievement goal $p(\vec{t})$, or a goal variable X, then $T_\pi = \langle E, \pi, E\rangle$,

(iii) If $\pi = \pi_1;\ \pi_2$ is a sequentially composed goal, then $T_\pi = \langle T_{\pi_1}, ;, T_{\pi_2}\rangle$,

(iv) If $\pi = \pi_1 + \pi_2$ is a choice goal, then $T_\pi = \langle T_{\pi_1}, +, T_{\pi_2}\rangle$,

(v) If $\pi = \pi_1 \| \pi_2$ is a parallel goal, then $T_\pi = \langle T_{\pi_1}, \|, T_{\pi_2}\rangle$.

The goal tree T_π corresponding to a goal π is called the *canonical representant* of π. Because we want to disregard the order of choice points and subgoals of parallel nodes in a goal tree, however, a goal may have more than one representant. All the representants of a goal are captured by the *forest of tree representants* of the goal which is derived by the operator r from the canonical tree representant.

Definition 4.3.2 *(forest of tree representants)*
The *forest of tree representants of* π, $r(T_\pi)$, is defined by:

- $r(\langle E, l, E\rangle) = \{\langle E, l, E\rangle\}$, where the root label l of the tree may be a basic action $\mathsf{a}(\vec{t})$, a test $\varphi?$, an achievement goal $p(\vec{t})$, or a goal variable X,
- $r(\langle T_{\pi_1}, ;, T_{\pi_2}\rangle) = \{\langle T_1, ;, T_2) \mid T_1 \in r(T_{\pi_1}), T_2 \in r(T_{\pi_2})\}$,
- $r(\langle T_{\pi_1}, +, T_{\pi_2}\rangle) = \{\langle T_1, +, T_2), \langle T_2, +, T_1\rangle \mid T_1 \in r(T_{\pi_1}), T_2 \in r(T_{\pi_2})\}$,
- $r(\langle T_{\pi_1}, \|, T_{\pi_2}\rangle) = \{\langle T_1, \|, T_2), \langle T_2, \|, T_1\rangle \mid T_1 \in r(T_{\pi_1}), T_2 \in r(T_{\pi_2})\}$.

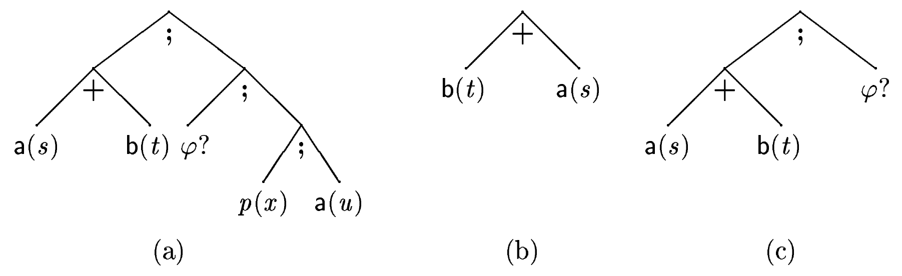

Figure 4.1: Three Goal Trees

In figure 4.1, the concept of a canonical representant is illustrated. Figure 4.1(a) depicts the canonical goal tree of the goal $(\mathsf{a}(s)+\mathsf{b}(t));\ (\varphi?;\ (p(x);\ \mathsf{a}(u)))$, figure 4.1(b) depicts the canonical representant of $\mathsf{b}(t) + \mathsf{a}(s)$ and, figure 4.1(c) canonically represents $(\mathsf{a}(s) + \mathsf{b}(t));\ \varphi?$.

A practical reasoning rule ***is applicable*** to a goal π if the head π_h of the rule *matches* with π and the guard φ of the rule is implied by the belief base of the agent. The notion of matching denoted by $\approx\!\mid$ in definition 3.6.1 is now formally defined by means of goal trees. Note that $\approx\!\mid$ is an a-symmetric relation.

Definition 4.3.3 *(definition of $\approx\!\!|$)*
Let π_1, π_2 be two semi-goals. Then:

$$\pi_1 \approx\!\!| \ \pi_2 \text{ iff } T_{\pi_2} \in r(T_{\pi_1})$$

By the definition of matching, a rule cannot distinguish the syntactic order of the two branches of a choice goal or subgoals of a parallel node. In other words, parallel composition and nondeterministic choice are commutative and thus the meaning of a goal $\pi_1 + \pi_2$, for example, is the same as that of $\pi_2 + \pi_1$. It is not just the case that these goals behave similarly, but they behave similarly *in any context.*

The concept of a context is defined in terms of semi-goals. A *context* is a semi-goal with exactly one occurrence of a goal variable. Thus, $\pi;\ X$ and $(\pi_1 + X);\ \pi_2$ are contexts if π, π_1, π_2 are goals. A context is written like $C[X]$ and the substitution of a semi-goal π for the goal variable X is written as $C[\pi]$. For example, if $C[X]$ is $(\pi_1 + X);\ \pi_2$, then $C[\pi]$ is $\pi_1 + \pi;\ \pi_2$.

A choice goal $\pi_1 + \pi_2$ thus is equivalent in any context to $\pi_2 + \pi_1$, and a goal $\pi_1 \| \pi_2$ is equivalent to $\pi_2 \| \pi_1$. This is formally stated in the next theorem. The techniques to prove this theorem and a definition of the notion of bisimilarity is presented in chapter 8. Informally, the theorem states that a goal $C[\pi_1 + \pi_2]$ and a goal $C[\pi_2 + \pi_1]$ or a goal $C[\pi_1 \| \pi_2]$ and a goal $C[\pi_2 \| \pi_1]$ produce exactly the same observable events and cannot be distinguished from each other.

Theorem 4.3.4 *(commutativity of nondeterministic choice)*
For any context, $C[X]$, $C[\pi_1 + \pi_2]$ is bisimilar to $C[\pi_2 + \pi_1]$ and $C[\pi_1 \| \pi_2]$ is bisimilar to $C[\pi_2 \| \pi_1]$.

Proof: By induction on the structure of the context $C[X]$. $\Box$

Since practical reasoning rules can modify programs, we cannot take basic properties such as that of the commutativity of nondeterministic choice for granted, and we need to prove them. Goal variables in practical reasoning rules allow an agent to modify its goals in almost any way. As a simple example of the range of possibilities, consider the rule $X;\ Y \leftarrow Y;\ X$. This rule swaps the order of two subgoals in a sequential goal and is called the *swap rule.* Although this rule may not be very useful, a special instantiation of the swap rule can be used to simulate parallelism as we will show below. The rule illustrates the type of modification that can be accomplished with practical reasoning rules. The effects of practical reasoning rules upon goals cannot in general be obtained by means of the other constructs available in the programming language (including simple plan rules without goal variables). It is, for example, not possible to simulate the swap rule by means of the IF...THEN...ELSE... construct. Practical reasoning rules thus add expressive power to the agent programming language.

One of the consequences of adding such general rules to the programming language, is that the classical law of the associativity of sequential composition no longer holds. That is, a goal $(\pi_1;\ \pi_2);\ \pi_3$ is not equivalent to the goal

$\pi_1;\ (\pi_2;\ \pi_3)$. This can be verified by applying the swap rule $X;\ Y \leftarrow Y;\ X$ to both goals. In the former case, the result is $\pi_3;\ (\pi_1;\ \pi_2)$, whereas in the latter case the result is $(\pi_2;\ \pi_3);\ \pi_1$. The order in which the subgoals can be executed in the first and in the latter case are different. Because the law of associativity does not hold, we stipulate, by convention, that sequential composition ; is *left associative.*[1]

A Least Commitment Approach

The definition of $\lessapprox\!\!|$ formalises the concept of matching that is used in the transition rule 3.6.1 for rule application. As we remarked throughout, however, a rule may be applied to a *subgoal* of a goal in the goal base of an agent. This is due to the *recursive decomposition* of a goal in a transition system.

The particular subgoals that may be modified by a practical reasoning rule are the subgoals that need to be taken care of first by the agent. In particular, a plan rule can only be applied to a goal with an achievement subgoal that must be achieved first to achieve the main goal. The decomposition in a transition system thus naturally supports a *least commitment approach* of agents. An agent does not make a commitment to a plan in case it does not have to make a choice at that same moment to make progress. Such an approach has the advantage that a plan is selected at a time that it is required and the most up-to-date information to select a plan is present. This minimises the likelihood that a revision is needed later.

Formally, the subgoals that can be modified by applying a rule are captured by the notion of a *left subtree.* Since a tree representation of a goal may be identified with a goal, we also speak about left subgoals. The head π_h of a rule is matched with a left subgoal of a goal in the goal base of the agent if it is applied. The notion of a left subtree is also related to the *control flow* of an imperative program. In imperative programming, always a left subprogram - a part 'at the front' - of a program is executed.

Definition 4.3.5 *(forest of left subtrees)*
Let T_π be a goal tree. Then: the *forest of left subtrees* derived from T_π, denoted by $\lambda(T_\pi)$, is defined by:

- $\lambda(\langle E, l, E\rangle) = \{\langle E, l, E\rangle\}$, where l denotes the label of the root node,
- $\lambda(\langle T_{\pi_1}, ;, T_{\pi_2}\rangle) = r(\langle T_{\pi_1}, ;, T_{\pi_2}\rangle) \cup \lambda(T_{\pi_1})$,
- $\lambda(\langle T_{\pi_1}, +, T_{\pi_2}\rangle) = r(\langle T_{\pi_1}, +, T_{\pi_2}\rangle) \cup \lambda(T_{\pi_1}) \cup \lambda(T_{\pi_2})$,

[1] Whereas the example in the text illustrates that due to practical reasoning rules the associativity of sequential composition fails, if each of the two programs $(\pi_1;\ \pi_2);\ \pi_3$ and $\pi_1;\ (\pi_2;\ \pi_3)$ is combined (only) with the swap rule, they are still operationally equivalent in the sense that they produce exactly the same computations. As an example in which this is not the case due to the presence of another practical reasoning rule, consider the two sequential goals a; (b; c) and (a; b); c, and the rule $(X;\ Y);\ Z \leftarrow (X;\ Z);\ Y$. Now, the former goal can only execute the sequence a, b, c, whereas the latter can also generate the sequence a, c, b.

- $\lambda(\langle T_{\pi_1}, \|, T_{\pi_2}\rangle) = r(\langle T_{\pi_1}, \|, T_{\pi_2}\rangle) \cup \lambda(T_{\pi_1}) \cup \lambda(T_{\pi_2})$.

$\lambda(T)$ is the *set of left subtrees* of a goal tree T, where the order of the branches branching from a node labelled with $+$ or $\|$ are disregarded. This explains why T_{π_1} and T_{π_2} are both considered to be left subtrees of $\langle T_{\pi_1}, +, T_{\pi_2}\rangle$ and $\langle T_{\pi_1}, \|, T_{\pi_2}\rangle$. The left subgoals (trees) are the possible candidates for modification of a goal (tree) at the agent level. By means of left subtrees, it is possible to define a notion of matching at the agent level. Let π_1 and π_2 be goals. Then π_1 is said to *match* with π_2 *at the agent level* if the goal tree T_{π_1} corresponding to π_1 matches with *a* tree in the forest $\lambda(T_{\pi_2})$.

As an illustration of matching, in figure 4.1, the goal $b(t) + a(s)$ matches with $(a(s) + b(t))$; $(\phi?$; $(P(x)$; $a(u)))$, but $(a(s) + b(t))$; $\phi?$ does not match with $(a(s) + b(t))$; $(\phi?$; $(P(x)$; $a(u))$.

Procedurally, the application of a rule ρ to a goal π in the goal base of an agent comes down to pruning a left subtree of π that matches with the head π_h of ρ, and replacing this subtree with the goal tree corresponding to the body π_b of ρ. In case some parent nodes of the relevant subtree are choice points, it is also necessary to remove certain other subtrees as a result of the choice (commitment) made. For example, if a rule matches with π_1 in $\pi_1 + \pi_2$, subgoal π_2 is also dropped when the rule is applied.

4.4 Posting Goals by Condition-Action Rules

Condition-action rules are a special type of rules different from other practical reasoning rules. This difference stems from the fact that these rules, unlike other rules, are not triggered by any current goals of the agent. Instead, these rules may be fired in case certain beliefs are present. If - based just on the beliefs of the agent - it is advantageous to adopt a plan of action, condition-action rules are the way to introduce such plans. Their special status among the other rules was already indicated by a special transition rule for condition-action rules.

A condition-action rule of the form $\leftarrow \phi \mid \pi$ is applicable at any time that condition ϕ is believed by the agent. ϕ thus may be called the *triggering condition* of the rule. If a condition-action rule is applied, a new goal is introduced into the goal base without any changes to the agent's beliefs. Since applying a condition-action rule does not affect the beliefs of the agent, the triggering condition of such rules thus is not disabled after their application. This is an important difference with other rules which replace a goal from the goal base and (may) disable their own triggering condition in that way. Because of this fact it is more difficult to control the application of condition-action rules.

There exists, however, a simple technique to gain more control over the application of condition-action rules. This technique consists of prefixing a disabling action to the body of such a rule that disables its triggering condition. Suppose, to illustrate, that a plan π is to be added to the goal base in case a condition ϕ is believed. A simple instance of the technique then consists in placing the action $\mathsf{del}(\phi)$ in front of the plan π resulting in the rule $\leftarrow \phi \mid$

del(ϕ); π. If the rule is fired, it inserts a plan into the goal base that disables its own triggering condition.

There remains, however, a theoretical problem. Even if the body of a condition-action rule may disable the triggering condition of that rule, it still may be the case that the condition-action rule is never disabled. This is a result of the fact that the disabling action in the plan may never be executed at all simply because it is never selected for execution. The operational semantics allows such 'unfair' computations. One example of such a computation is the computation that consists of forever repeating the transition in which the condition-action rule is applied. In this example computation, the application of the relevant condition-action rule is favoured over all other possible transitions (for example, removing the triggering condition by the del action). This problem of computations 'preferring' certain transitions over others is called the *fairness problem* (cf. Manna & Pnueli (1992)). To solve this problem, we need to exclude such *unfair* computations. For this purpose, we introduce a so-called *weak fairness condition* (on transitions):

it is not allowed that a transition is continuously enabled from some time on in a computation, but that same transition is never taken during the computation.

It is not too difficult to see that if computations obey this fairness rule, computations such as the example computation in which nothing else happens than the application of condition-action rules is excluded.

Posting Goals

As an illustration of the disabling technique, we show how an operator post(π) to post a new goal π can be implemented in 3APL by means of condition-action rules. First, we introduce new transition rules which make precise what the post construct is supposed to do. Then we propose an implementation of this operator.

Informally, the action post(π) adds the goal π to the goal base of the agent. The action itself is an action that is part of a goal. The execution of actions within goals was dealt with in chapter 3 at the goal level. However, the action of posting a goal π at the *goal level* must be accounted for at the *agent level* since the goal π must be added to the goal base of the agent. In this case, however, we cannot introduce a transition rule at the agent level for dealing with the post action as we did for condition-action rules. The point is that a post action is a subgoal in one of the goals in the goal base and at the agent level we are not able to directly inspect subgoals.

An elegant way to pass on the information that a goal has been posted by executing an action at the goal level to the agent level is to use labels. A special set of labels of the form post(π) is introduced to distinguish transitions in which a goal π has been posted from other transitions. Since label information is passed on from the goal to the agent level, the information that a goal has been posted and also which goal has been posted then is also available at that level.

Definition 4.4.1 *(transition rule for* post *at the goal level)*
Let π be a *closed* goal, that is, $\mathsf{Free}(\pi) = \varnothing$.

$$\frac{}{\langle \mathsf{post}(\pi), \sigma\rangle_V \xrightarrow{\mathsf{post}(\pi)}_{\varnothing} \langle E, \sigma\rangle}$$

The goal π that is posted is required to be closed *upon execution of the* post *operator*. $\mathsf{post}(\pi)$ may contain free variables initially, which are instantiated with appropriate values during the execution of the agent. The requirement that π must be closed derives from the requirement that all basic actions are closed upon execution. The fact that π must be closed upon execution is not a real restriction, however, since π may contain achievement goals which may introduce new local variables. Moreover, the only type of communication between goals proceeds by means of the belief base and not by means of shared variables.

At the agent level, a distinction needs to be made now between transitions with special post labels and other labels. To this end, we need to slightly modify the semantics of the agent level of definition 3.4.2. In case a label different from a post label is associated with a transition at the goal level, this label is passed on to the agent level exactly as before. However, if a $\mathsf{post}(\pi)$ label is associated with a transition at the goal level, the goal π is added to the goal base and a τ-label is associated with the agent level. The τ-label indicates that an action internal to the agent has been performed.[2]

Definition 4.4.2 *(transition rules for* post *at the agent level)*
Let $V = \mathsf{Free}(\pi_i)$ and let θ be a substitution.

$$\frac{\langle \pi_i, \sigma\rangle_V \xrightarrow{l}_\theta \langle \pi_i', \sigma'\rangle, l \neq \mathsf{post}(\pi)}{\langle\{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots, \pi_n\}, \sigma\rangle \xrightarrow{l} \langle\{\pi_0, \ldots, \pi_{i-1}, \pi_i', \pi_{i+1}, \ldots, \pi_n\}, \sigma'\rangle}$$

$$\frac{\langle \pi_i, \sigma\rangle_V \xrightarrow{\mathsf{post}(\pi)}_{\varnothing} \langle \pi_i', \sigma'\rangle}{\langle\{\pi_0, \ldots, \pi_{i-1}, \pi_i, \pi_{i+1}, \ldots, \pi_n\}, \sigma\rangle \xrightarrow{\tau} \langle\{\pi_0, \ldots, \pi_{i-1}, \pi_i', \pi_{i+1}, \ldots, \pi_n, \pi\}, \sigma'\rangle}$$

Implementing post **with Condition-Action Rules**
To implement $\mathsf{post}(\pi)$ by means of other facilities in the agent language, we need to find a plan and/or a set of rules which can be substituted for the action

[2]The combination with the semantics of priorities introduced in definition 4.2.1 is quite straightforward, and the details are not spelled out here. However, now we have introduced a new set of labels $\mathsf{post}(\pi)$, we should also discuss its place in the priority ordering. Because the post action is viewed here as a regular action, these labels are assigned the same priority as any other action.

$\mathsf{post}(\pi)$ and which has the same effects. The basic idea is to use a condition-action rule to add the goal that is posted to the goal base. Since a goal should only be added to the goal base if a post action has been executed, the post action should be implemented by a plan which enables the triggering condition of the rule. For this purpose, a special predicate $\mathit{create}_\pi(\vec{x})$ is introduced which is used as the triggering condition of the rule where $\vec{x}$ are the free variables (initially) in the goal $\pi(\vec{x})$. It is assumed that this predicate is not used for any other purposes by the agent.

The main job of the plan to implement the $\mathsf{post}(\pi(\vec{x}))$ action thus is to establish the triggering condition of the rule. To this end, we use the action $\mathsf{ins}(\mathit{create}_\pi(\vec{x}))$ which inserts this condition into the belief base.[3] To make sure that only one post action is executed at the time, binary semaphores s_π are used to guarantee mutual exclusion with other parallel sections which implement the same $\mathsf{post}(\pi)$ action. Binary semaphores are formally introduced and defined in chapter 6. Informally, semaphores are constructs that can be used to ensure that two parallel processes do not execute an action simultaneously. Finally, the function of the test in the goal guarantees that the agent exits the critical section only after the associated triggering condition has been removed again. The plan that is substituted for $\mathsf{post}(\pi(\vec{x}))$ then is the following goal:

(1) $\quad \mathbf{P}(s_\pi);\ \mathsf{ins}(\mathit{create}_\pi(\vec{x}));\ \neg\mathit{create}_\pi(\vec{x})?;\ \mathbf{V}(s_\pi)$

The triggering condition of the condition-action rule to create the goal $\pi(\vec{x})$ is the special predicate $\mathit{create}_\pi(\vec{x})$. The disabling technique discussed previously is used to make sure that the rule introduces at most one copy of the posted goal into the goal base. The triggering condition can be disabled by deleting the predicate $\mathit{create}_\pi(\vec{x})$ by a del action prefixed to the posted goal $\pi(\vec{x})$. The rule for posting $\pi(\vec{x})$ then is the following condition-action rule:

(2) $\quad \leftarrow \mathit{create}_\pi(\vec{x}) \mid \mathsf{del}(\mathit{create}_\pi(\vec{x}));\ \pi(\vec{x})$

Summarising, the goal (1), which replaces the $\mathsf{post}(\pi)$ goal, inserts a predicate create_π in the belief base of the agent which triggers the condition of the condition-action rule (2). After inserting the trigger create_π in the belief base, the goal (1) waits until the trigger is removed again. The condition-action rule (2) eventually will fire because its trigger condition is true.[4] Upon firing, the rule creates a goal $\mathsf{del}(\mathit{create}_\pi);\ \pi$. Eventually, the $\mathsf{del}(\mathit{create}_\pi)$ action will be executed which removes the triggering condition and results in a goal base containing π. After the triggering condition is removed, the agent can exit the critical section of the plan of rule (2) and execution continues as if a post action has been performed.

[3]Because we assume that the predicate $\mathit{create}_\pi(\vec{x})$ is not used anywhere else, the insertion of this proposition into the belief base by $\mathsf{ins}(\mathit{create}_\pi(\vec{x}))$ cannot result in consistency conflicts with things already believed by the agent and thus will always succeed.

[4]And because weak fairness is assumed and the fact that only the body of the rule (2) removes the triggering condition.

4.5 Failure, Recovery and Monitoring in 3APL

A facility for dealing with failure and the recovery from failure is provided for by failure rules in 3APL. A facility for dealing with failure thus is an integral part of the agent language and allows for various ways of dealing with failure that can be *programmed.* In most agent frameworks, it is not so easy to integrate these abilities in a clean and flexible way. In most cases, these facilities are statically built into the control structure of agents (for example, an agent always tries an alternative plan to achieve a goal when a previously adopted plan fails, as is the case in dMars (d'Inverno et al. 1998), whereas in AGENT0 an agent always removes actions from its commitment base if it is not able to perform them given its beliefs in the current situation (Shoham 1993)).

In this section, we investigate failure rules and techniques for using them. Although in section 4.1 a general outline of some strategies for dealing with failure was presented, this outline only discussed the uses of failure rules. It did not yet provide concrete techniques for programming with failure rules. A technique called *postfix-labelling* is introduced to gain more control over the application of failure rules. Then we show that by using this technique we are able to implement two types of monitoring facilities by means of failure rules. The first facility is a so-called *disruptor mechanism* and the second facility is an *interrupt mechanism.* The implementation of these mechanisms shows that failure rules in 3APL provide a natural and flexible mechanism for implementing failure facilities.

4.5.1 Postfix-Labelling

Practical reasoning rules allow the modification of goals of agents in almost arbitrary ways. This power to modify goals in arbitrary ways partly stems from the use of goal variables that range over arbitrary goals. As before, with condition-action rules, however, it is not always easy to control the application of practical reasoning rules with arbitrary (non-empty) head. A striking example is again provided by the swap rule $X;\ Y \leftarrow Y;\ X$. This rule applies to *any* sequential goal and reverses the order of its subgoals. There are few, if any at all, situations in which such a rule would be useful. The problem with rules like the swap rule is that they *are not focused.* It is therefore difficult to control the application of such rules.

A technique called *postfix-labelling* is introduced to gain more control over the application of practical reasoning rules with goal variables in the head of the rule. The idea is to mark any goal that might be modified by a rule with goal variables with a unique *label.* By labelling the relevant goal variables in a similar way, these variables then are only applicable to these labelled goals. Of course, we do not want this labelling to affect computation steps in which no rule (or other rules) is (are) applied. For this reason, the label should be associated with the goal in a particular way to avoid such interference. It turns out that *postfixing* the label to the goal that is to be labelled solves the problem. That is, if a (sub)goal π is to be labelled with a label l, then the label

l is sequentially composed with the original (sub)goal π and we obtain the new labelled (sub)goal $\pi;\ l$. Similarly, goal variables X that should match with the (sub)goal are labelled with l and we need to replace X with $X;\ l$.

A set of labels L thus is introduced into the agent language as a way to control the application of practical reasoning rules with variables. These labels are part of the agent language itself and should not be confused with labels associated with transitions or with labels associated with the nodes of a goal tree. Since these labels are part of the language we need to define their semantics. Of course, labels should not have any effect on the beliefs of the agent and also should not result in deadlock. In fact, the 'execution' of labels is defined as that of ***skip*** actions. When they are encountered during the computation, they are simply removed and execution continues as if no label was present. A transition rule at the goal level for labels formalises this semantics. A τ-label is associated with a transition for a label since it is considered an internal step.

Definition 4.5.1 *(labels)*
Let $l \in L$ be a label.

$$\frac{}{\langle l, \sigma\rangle \xrightarrow{\tau}_{\varnothing} \langle E, \sigma\rangle}$$

To illustrate the technique of postfix-labelling, consider figure 4.2. Suppose the subgoal π_2 in the goal $(\pi_1;\ \pi_2);\ \pi_3$ needs to be singled out as a goal that should match with a goal variable in a practical reasoning rule. In that case, we say that the goal π_2 is *monitored* by the rule. The postfix-labelling technique is used to guarantee that the goal variable matches with the goal to be monitored. A label l is introduced that is not used anywhere else within the agent and π_2 is replaced with $\pi_2;\ l$. This replacement is illustrated in figure 4.2. The subtree corresponding to the subgoal π_2 is removed and the goal tree for $\pi_2;\ l$ is substituted in its place. Somewhat more formally, let $C[X] = (\pi_1;\ X);\ \pi_3$ be the context such that $C[\pi_2] = (\pi_1;\ \pi_2);\ \pi_3$. Postfix-labelling π_2 with l in the latter goal then means that $\pi_2;\ l$ is inserted in the context $C[X]$ instead of π_2 resulting in the goal $(\pi_1;\ (\pi_2;\ l));\ \pi_3$.

Similarly, every occurrence of the goal variable X in the rule is replaced with its labelled variant $X;\ l$. As a simple example, consider the rule $X;\ l \leftarrow \varphi$ that drops a goal labelled with l in case φ holds and the goal $(\pi_1;\ (\pi_2;\ l));\ \pi_3$. Clearly, this rule is only applicable after π_1 has been done and $(\pi_2;\ l);\ \pi_3$ is the current goal. During the execution of π_2, however, the head of the rule still matches with the subgoal. So, if π_2 has been transformed to π_2', then the rule may be applied to the new goal $(\pi_2';\ l);\ \pi_3$ in case the guard φ holds and π_2' may be dropped. The goal π_2 thus is continually monitored during its execution.

The labelling technique allows us to control the application of practical reasoning rules with far more precision. The technique, however, does not always work in case labelled and unlabelled goal variables are mixed and used simultaneously. To see what might go wrong, again consider the swap rule $X;\ Y \leftarrow Y;\ X$. The labelling technique is based on the assumption that a label l is *postfixed* to a goal π resulting in $\pi;\ l$. However, the swap rule may

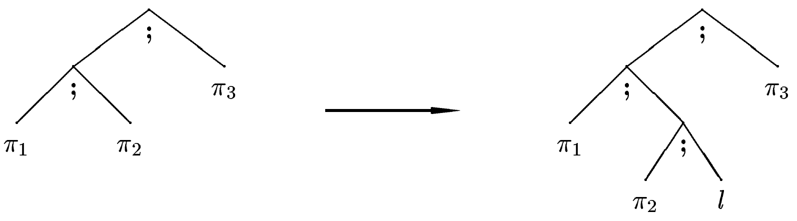

Figure 4.2: Postfix-labelling

swap the order of the label and the goal and *prefix* the label to the goal. Other types of rules with unlabelled goal variables might simply drop labels or mix up things in yet other ways.

To prevent such interactions between labelled and unlabelled goal variables, we therefore will assume that labelled goals and goal variables are taken as a single construct. That is, a label in a statement $X;\ l$ or $\pi;\ l$ from now on should be considered as a single and new syntactic category. To emphasise this, we introduce some notation for labelling goals and goal variables. A *labelled goal variable* $X;\ l$ is from now on written as X_l; a labelled goal $\pi;\ l$ is written as $[\pi]_l$. This notation is introduced as a uniform notation for labelling; the semantics of labels as specified by definition 4.5.1 can be copied and used to define the semantics of both labelled goal variables X_l as well as labelled goals $[\pi]_l$. The notation suggests that labelling is similar to a kind of typing mechanism.

4.5.2 Disruptors

To illustrate the use of labelled goal variables and failure rules, we show how to implement two facilities for monitoring goals: a *disruptor* and an *interrupt*. A *disruptor* $\mathsf{disrupt}(\phi, \pi_m, \pi_r)$ is a facility that monitors a particular goal π_m. π_m is called the *monitored goal* and is executed by the agent in normal circumstances. However, when the associated *monitoring condition* ϕ holds, execution continues with the so-called *recovery goal* π_r. In that case, the monitored goal π_m is replaced with the recovery goal π_r. In case of failure, a disruptor typically is used to implement the strategy of re-planning from scratch. The informal discussion of the operation of the disrupt construct is made formal by the following transition rules at the goal level. $\mathsf{disrupt}(\phi, E, \pi_r)$ is identified with E. Because a disruptor is here primarily taken as a facility for dealing with failure, the f label (for failure rules) is associated with a computation step of a disruptor (the second transition rule below). In case the monitored goal is executed, the label is the same as that of the computation step of that goal.

Definition 4.5.2 *(disruptor mechanism)*

$$\frac{\langle \pi_m, \sigma\rangle \xrightarrow{l}_\theta \langle \pi'_m, \sigma'\rangle \text{ and } \forall\gamma : \sigma \not\models \phi\gamma}{\langle \mathsf{disrupt}(\phi, \pi_m, \pi_r), \sigma\rangle \xrightarrow{l}_\theta \langle \mathsf{disrupt}(\phi, \pi'_m, \pi_r), \sigma'\rangle}$$

$$\frac{\sigma \models \phi\theta, dom(\theta) = \mathsf{Free}(\varphi)}{\langle \mathsf{disrupt}(\phi, \pi_m, \pi_r), \sigma\rangle \xrightarrow{f}_\theta \langle \pi_r\theta, \sigma\rangle}$$

The transition rules cover all cases. That is, either the monitoring condition ϕ is believed by the agent and we have $\sigma \models \phi$ or it is not believed and we have $\sigma \not\models \phi$. A disruptor thus never deadlocks like an IF ... THEN ... ELSE .... The monitoring condition determines when the disrupt should be executed and the recovery goal should replace the monitored goal. In case the monitoring condition does not hold, the monitored goal π_m is executed just as it would have been without the presence of the disruptor. The first transition rule formalises this case. A disruptor remains present during the normal execution of the monitored goal and is removed only when the monitored plan has been finished or the recovery goal is substituted for it. This means that all goals resulting from executing one or more steps of the original goal π_m thus are also monitored. In case the agent believes that ϕ, the second transition rule formalises that π_m is replaced with the recovery goal π_r and execution continues with the recovery goal π_r as the main goal. The monitoring condition thus specifies in what circumstances the plan π_r should take over.

Implementation of disruptors

The technique of postfix-labelling will now be used to implement disruptors by means of practical reasoning rules. A practical reasoning rule that implements a disruptor is called a *disruptor rule*. The implementation of a disruptor proceeds by a straightforward application of the labelling technique. To guarantee that the monitored goal only is affected by the disruptor rule that implements the disruptor, labels are introduced to uniquely identify the monitored goal. To implement a disruptor $\mathsf{disrupt}(\phi, \pi_m, \pi_r)$ in a context $C[X]$, then, we need to do two things: (i) label the monitored goal π_m by a unique label l and substitute the labelled goal $[\pi_m]_l$ in the context $C[X]$, and (ii) create a disruptor rule $X_l \leftarrow \phi \mid \pi_r$ where ϕ is the monitoring condition of the disruptor and π_r is the associated recovery goal. A disruptor rule is applicable just in case the associated disruptor replaces the goal π_m by π_r, i.e. ϕ holds.

Postfix-labelling guarantees that a disruptor rule *is applicable* to the monitored goal if and only if the monitoring condition holds. It does not imply, however, that the rule *is applied* in case the monitoring condition holds, nor that the monitored goal is no longer executed. To guarantee similar behaviour of the disruptor rule and the disruptor, a preference for applying disruptor rules over action execution must be imposed. For this purpose, we can use the classification of practical reasoning rules and the associated priorities. The formal semantics was defined by the new agent level transition rule 4.2.1. By classifying disruptor rules as failure rules, they are preferred over action execution. Consequently, disruptor rules implement disruptors in the semantics with priorities of definition 4.2.1. Notice that in this case we did not need the full structure of the classification into four classes. A simpler scheme of priorities would have been sufficient here.

4.5.3 Interrupts

Another facility for monitoring goals are *interrupts* $\mathsf{interrupt}(\phi, \pi_m, \pi_i)$. As before, the goal to be monitored is π_m, and ϕ is the monitoring condition. Instead of replacing the monitored goal with another goal, however, an interrupt mechanism interrupts the goal π_m and first executes the *interrupt goal* π_i. This type of monitoring facility can be used, for example, for monitoring preconditions of an adopted plan. In case a precondition of such a plan fails to hold, an interrupt goal can be invoked to attempt to re-establish the preconditions of the plan. The formal operation of interrupts is defined by the following transition rules. Again, $\mathsf{interrupt}(\phi, E, \pi_i)$ is identified with E. The labelling of transitions is identical to that of disruptors.

Definition 4.5.3 *(interrupt mechanism)*

$$\frac{\langle \pi_m, \sigma\rangle \xrightarrow{l}_\theta \langle \pi'_m, \sigma'\rangle \text{ and } \forall\, \gamma : \sigma \not\models \phi\gamma}{\langle \mathsf{interrupt}(\phi, \pi_m, \pi_i), \sigma\rangle \xrightarrow{l} \langle \mathsf{interrupt}(\phi, \pi'_m, \pi_i), \sigma'\rangle}$$

$$\frac{\sigma \models \phi\theta,\, dom(\theta) = \mathsf{Free}(\varphi)}{\langle \mathsf{interrupt}(\phi, \pi_m, \pi_i), \sigma\rangle \xrightarrow{f}_\theta \langle \pi_i\theta;\ \mathsf{interrupt}(\phi, \pi_m, \pi_i), \sigma\rangle}$$

Implementation of Interrupts

The implementation of interrupts by practical reasoning rules is almost completely analogous to that of disruptors. The goal that is to be monitored is labelled and a practical reasoning rule that is called an *interrupt rule* is introduced. An interrupt rule is of the same form as a disruptor rule except for the body. Where the body of a disruptor rule simply consists of the recovery goal that replaces the monitored goal, the body of an interrupt rule inserts the interrupt goal into the goal base but does not drop the monitored goal. The body of an interrupt rule then becomes π_i; $([\pi_m]_l)$ which guarantees execution of the interrupt goal first and then continues with the same interrupt as before. Thus, to implement an interrupt $\mathsf{interrupt}(\phi, \pi_m, \pi_i)$ in a context $C[X]$, two things have to be done: (i) label the monitored goal π_m by a unique label l and substitute the labelled goal $[\pi_m]_l$ in the context $C[X]$, and (ii) create an *interrupt rule* $X_l \leftarrow \phi \mid \pi_i;\ X_l$. As in the case of disruptors, interrupt rules are classified as failure rules.

Two types of monitoring mechanisms have been implemented by means of practical reasoning rules. In both cases, a label was introduced to monitor a single goal. The use of labels, however, is not restricted to the application of a single goal. It is also possible, for example, to use a single label to monitor two parallel goals labelled with the same label simultaneously. Still other applications are conceivable, where a single label is used to label more than one goal variable. An example of the latter is provided in the next section.

4.6 The Swap Rule and the Simulation of Parallelism

To illustrate the expressive power of practical reasoning rules, we discuss one more example. The aim is to show that practical reasoning rules can be used to simulate parallelism. To this end, we must show that for arbitrary agents we can always find another agent that behaves similarly and which does not have multiple goals nor uses parallel operators in its plans.

To achieve our goal, we must first gain a better understanding of the semantics of parallelism. First of all, the parallelism that we are interested in here is the implicit parallelism of multiple goals in the goal base of an agent and not the parallelism of parallel goals $\pi_1 \| \pi_2$. This type of parallelism is modelled by an *interleaving semantics* in the operational semantics. It is thus important to understand this model. Interleaving semantics imposes a particular schedule on top of the execution of multiple goals. It allows one goal at a time to perform a computation step. This type of semantics thus suggests that a scheduler each 'tick of the clock' selects a goal that may execute and perform a computation step of that goal. The scheduler thus determines the *order* in which the goals are executed and generates a particular *sequence of actions* that is performed. The selection of a goal in an interleaving semantics, however, should be arbitrary. That is, the scheduler may not prefer the execution of any specific goal over that of others.

The key to simulating parallelism thus is to generate all the possible execution sequences of actions of parallel goals. One such a sequence is, of course, the sequential composition of all the parallel goals. This is only one possible sequence, however, and does not allow for alternative action sequences in which actions of goals later in the sequential composition are executed before those of goals earlier in the sequence. But, by now, we know that the order of individual goals in a sequence of goals can be changed by practical reasoning rules. As an example, we have cited the swap rule more than once. The idea is to simulate the parallel execution of n goals by putting these goals in a sequence and by using the swap rule to change the order of the goals in the sequence. By arbitrarily swapping the order of the goals in the sequence it is possible to execute actions from any goal. The resulting sequences of actions then is the same as that of an interleaving semantics.

To implement the idea for simulating parallelism, we use the postfix-labelling technique to identify the parallel goals and we use 'labelled' versions of the general swap rule. Let $\pi_1, \ldots, \pi_n$ be a set of goals of an agent. These goals are executed in parallel. The idea is to put these goals in a sequence, and to use a label to identify the goals that are executed in parallel. For this purpose, we use a single *:exit* label to indicate the end of a goal π_i. The n original goals then are labelled and sequentially composed into the new goal $((\ldots([\pi_1]_{:exit};\ [\pi_2]_{:exit});\ \ldots);\ [\pi_n]_{:exit})$. Since there is no communication between parallel goals, we assume that the sets of first order variables in the different goals are disjoint.[5]

The next step is to introduce practical reasoning rules that change the order of the goals in the sequence. The required rules must be able to swap goals and put an arbitrary goal first. For this purpose, we introduce two kinds of rules. Both rules are 'labelled versions' of the general swap rule $X;\ Y \leftarrow Y;\ X$. The first swap rule is $X;\ (Y_{:exit}) \leftarrow (Y_{:exit});\ X$ and is used to put an arbitrary goal $Y_{:exit}$ from the sequence in first place. The second swap rule is $(X_{:exit});\ Y \leftarrow Y;\ (X_{:exit})$ and is used to swap a goal $X_{:exit}$ at the front of the sequence with a subsequence Y immediately following the goal. The two rules are inverses of each other: the effects of either one of the rules can be cancelled by the other.

From our previous discussion, it will be clear what happens when this new agent is executed. By applying the swap rules an arbitrary goal can become first in the sequence of goals. Any goal that is first in the sequence can execute its actions, and as a result we obtain the arbitrary interleaving of actions from the goals.

There remain some issues that need to be settled. As before, we still need to assign the labelled swap rules to priority classes. Since in this case, the application of swap rules should not have priority over action execution, the swap rules are classified as optimisation rules. In the semantic framework with priorities, then, neither is preferred over the other. Furthermore, we need the weak fairness assumption for the simulation agent to work properly. The assumption is needed to exclude computations which continuously apply the swap rules and do nothing else. Finally, there is a problem concerning condition-action rules. Condition-action rules introduce new parallel goals, whereas in the simulation agent the assumption is made that such goals occur somewhere in the sequence of goals labelled with *:exit*. In this context, therefore, we must assume that condition-action rules are absent.

4.7 Conclusion

The practical reasoning rules of 3APL agents provide a powerful mechanism for goal and plan revision. In this chapter, we studied both the technical details involved in applying practical reasoning rules as well as the range of application of these rules. The former resulted in a formal definition of the *matching* of a rule and a goal. The broad range of uses of these rules was illustrated in several ways. A classification of practical reasoning rules in different rule classes was proposed as a guide to their use. Four classes were identified: failure rules, plan rules, condition-action rules and optimisation rules. The expressive power of rules was illustrated by an application of two types of rules. Condition-action rules can be used to implement a post operator for the creation of new goals. A technique for disabling the triggering conditions of such rules was discussed. Failure rules can be used to implement different monitoring facilities. Two such facilities were implemented: disruptors and interrupts. A technique

[5]The simulation of parallel goals thus is modulo the naming of first order variables. This is a technical detail that we do not go into here any further.

called postfix-labelling was introduced to gain more control over the application of rules with goal variables. The combination of practical reasoning rules and priorities turns out to be especially worth while. Priorities that are associated with computation steps provide yet another means to increase the control over the application of rules. Without priorities it would not have been possible to implement disruptors or interrupts. Finally, as yet another illustration of the expressive power of practical reasoning rules, it is possible to simulate parallelism by means of rules.

CHAPTER 5

Communicating Agents

In previous chapters, an intelligent agent has been defined as a computational entity consisting of beliefs and goals which make up its mental state. A semantics for multi-agent systems was also introduced. So far, however, we have only discussed intra-agent communication - that is, communication at the goal and agent level. Communication between agents - at the multi-agent level - has not yet been introduced. In this chapter, we extend the agent language 3APL with communication at the multi-agent level. Communication at the multi-agent level consists of communication between agents of their beliefs and goals. Two pairs of communication primitives are introduced for inter-agent communication. The semantics of these primitives is based on two distinct types of reasoning: *deduction* and *abduction*. The focus in the semantics is on the agent that is receiving a message, also called the *hearer*. Deduction allows the hearer to derive information from a received message. Abduction allows the hearer to obtain appropriate proposals in reply to requests. Our concern with the hearer differs from the standard speech act approaches in the literature.

5.1 Communication at the Multi-Agent Level

At the multi-agent level, a new concern of how to manage the interaction of the agents in a multi-agent system arises. One of the ways to structure the interaction of a set of agents is to use an *agent communication language* (another possibility would be to equip agents with sensory abilities). Communication allows agents to form teams to cooperate, or to negotiate to resolve conflicts. Communication involves two (or more) agents which interact by exchanging messages. Our focus will be mainly on the *agent that receives a message*. It is the agent which receives a message that has to process this message, and in the semantics for the communication language we want to capture this aspect of communication. Our approach thus contrasts with other approaches based on speech act theory, since traditionally speech act theory (Searle 1969) has been more speaker-oriented.

The fact that our main concern is with the hearer does not exclude the possibility to deal with other aspects of communication. Constraints on the speaker, for example, can be *programmed* in the agent programming language. Since such aspects can be programmed by using other facilities of the agent language, we do not need to incorporate them in the semantics of the communication primitives.

In this chapter, we discuss two types of message exchange. First, we discuss the exchange of a message which is interpreted as *providing an answer to a question.* Secondly, we discuss the exchange of a message which is interpreted as *making a request.* For both types of messages we propose a semantics which we believe captures the 'successful' processing of these messages by the receiver. In the following sections, we will explain what we mean by the *successful processing* of a message. Although we do provide a specific semantics in the formal definitions below for both types of communication, we want to emphasise that these definitions are particular instances of a more general idea: to base a semantics for the exchange of information on deductive reasoning and to base a semantics for the exchange of a request on abductive reasoning. The specific definitions below are illustrations of these ideas. To highlight the fact that a semantics based on deduction and abduction specifies a range of alternatives, we discuss some of the more interesting alternatives to the formal definitions of the semantics.

The exchange of messages between two agents involves the sending of a piece of information by one of the agents and the receiving of that information by the other agent. The language that is used to communicate messages is the knowledge representation language of the agent system. In the context of communication, this language is also called the *content language* and we assume that this language is a language shared by all agents in a multi-agent system. Corresponding to the two types of message exchange, we introduce two *pairs of communication primitives.* The first set of communication primitives, tell and ask, may be used for the exchange of information. A communicative action $\mathsf{tell}(b, \varphi)$ is used by the sender to communicate the message φ to a receiving agent b. A communicative action $\mathsf{ask}(a, \psi)$ is used by the receiver and specifies what type of information a corresponding message from a should provide in order to answer the question ψ. The second set of communication primitives, req and offer, may be used for the exchange of a request. A communicative action $\mathsf{req}(b, \varphi)$ is used by the sender to communicate a request φ to a receiving agent b. A communicative action $\mathsf{offer}(a, \psi)$ is used by the receiver and specifies a proposal ψ that the agent is prepared to make to satisfy a request of agent a.

5.2 A Formal Semantics for Communication

As before, a formal operational semantics for communication is defined by a number of transition rules. The formal semantics for the communication primitives that will be defined in the sequel is similar in a number of respects to that of CSP (Hoare 1985) and CCS (Milner 1989). The main difference is that our primitives concern the communication of messages from a *logical language*, instead of values that are associated with variables. A similar technique is used to specify the semantics, however. First, labelled transition rules are introduced for the individual communicative actions, like tell and ask, for example. The labels associated with these transitions are used to indicate that an *attempt* to communicate was made. Actual communication then takes place if labels of a sending agent and a receiving agent match.

This type of semantics specifies *synchronous* communication primitives. Synchronous communication of messages involves the participation of both agents at the same time and implies that the agents have to synchronise their communicative actions. The synchronisation of communicative actions gives rise to a simultaneous communicative act of two agents which is also called a *handshake*. A handshake is an atomic act, in the sense that it cannot be interrupted by other actions. Although synchronous communication primitives may seem to have a number of disadvantages compared with asynchronous communication primitives, computationally synchronous communication is as expressive as asynchronous communication. Asynchronous communication between two agents can be simulated by synchronous communication and a third 'interface agent' which behaves like a buffer. The idea is that an interface agent is always ready to receive messages from other agents (and therefore no delay in waiting for synchronisation occurs) and that an interface agent sends a message that it received for another agent whenever that agent is ready to receive the message. Moreover, we believe that for some purposes it is quite natural to have synchronous communication at the multi-agent level. The implementation of a multi-stage negotiation protocol in the next chapter provides an illustration of the use of synchronisation.

The use of synchronous communication between agents implies that a question and an answer, or a request and an offer, are exchanged 'simultaneously' by the agents involved in the communication. We call the agent that tells something to or requests something from another agent the *sender*, and the other agent that replies with an answer or an offer is called the *receiver*. In the sequel, we will see that this is a natural convention corresponding with the computational burden on the receiver to 'decode' the message that is received.

5.3 Information Exchange

We first introduce two communication primitives $\mathsf{tell}(b, \varphi)$ and $\mathsf{ask}(a, \psi)$ for the exchange of information. As mentioned previously, in the semantics we want to capture the *successful processing of a message by the receiver*. In the case of information exchange, from the point of view of the receiving agent, we think it is most important that that agent is able to *derive an answer to a question from the message* it received. Therefore, the successful processing of the message is identified with the process of deriving an answer from that message. Because messages and questions are formulas from the same logical language, it is most natural to process a received message by deriving an answer using *logical deduction*. The formal semantics for information exchange therefore is based on *deduction*.

We want to emphasise that our semantics is not intended to capture the much more complicated semantics of a speech act like, for example, informing. Whereas speech act theory is mainly concerned with constraints on the mental state of the speaker, we are interested in the way the hearer processes the received message.

Informing Another Agent

The communicative action $\mathsf{tell}(b, \varphi)$ is an action to send the message φ to agent b. There are *no* constraints on the sending agent imposed by the semantics; the only constraint is that agent b, to whom the message is sent, 'accepts' the message. The informal interpretation we associate with the action $\mathsf{tell}(b, \varphi)$ is that φ conveys some *information*.

The message φ is required to be a sentence *when* $\mathsf{tell}(b, \varphi)$ *is executed*, since it does not make sense to send indefinite information to another agent when an attempt is made to inform that agent. So, any free variables which occur in a $\mathsf{tell}(b, \varphi)$ action in a plan of an agent must have been instantiated by retrieving bindings for these variables and parameter passing before the formula φ can be communicated. The empty substitution $\varnothing$ is associated with the transition of tell since the sender receives no information. The transition below is labelled with $b!_i\varphi$ to indicate that an attempt of *sending* information φ to agent b is being made. The exclamation mark ! indicates that a message is being sent, while the subscript i indicates the type of communication, namely information exchange.

Definition 5.3.1 *(transition rule for* tell*)*

$$\frac{\varphi \text{ is closed}}{\langle \mathsf{tell}(b, \varphi), \sigma\rangle_V \xrightarrow{b!_i\varphi}_{\varnothing} \langle E, \sigma\rangle}$$

As explained in the previous section, the transition rule for tell and the transition rule for ask below specify *virtual* or *attempted* computation steps and *do not* specify completed computation steps. Actual communication between agents only occurs when the labels associated with the virtual steps match and the message sent is accepted by the receiving agent; in that case, a handshake occurs. The precise conditions for actual communication are specified in the third transition rule for the exchange of information below.

Asking for Information

The communicative action $\mathsf{ask}(a, \psi)$ is an action which specifies a condition for the acceptance of messages sent by a. The condition is that the message sent must entail (an instance of) ψ. The informal interpretation we associate with this action is that ψ is a *question* for which agent a should provide an answer.

The question ψ does not have to be a sentence, but may contain free variables. The free variables in ψ indicate what the question is about (cf. also Groenendijk & Stokhof (1984)). The receiving agent has to process a received message to compute an answer from that message. The process involved, as we suggested, is that of deduction. The receiving agent attempts to *deduce* an *answer* θ from a message φ of the sender a and its own beliefs σ. Recall that an *answer* to a question is a substitution θ such that $\sigma \cup \varphi \models \psi\theta$ (cf. definition 3.3). Moreover, we require that θ is a *complete* answer, that is, grounds ψ. The receiving agent accepts a message φ only if it is able to compute a complete answer to its question from φ.

In the transition rule for ask, the question mark ? in the label indicates that the asking agent is the receiving agent. The subscript i in the label indicates that the communication concerns information exchange. The substitution θ denotes a possible answer. The fact that there are no restrictions imposed on answers by the transition rule indicates that from the perspective of the receiving agent any answer is acceptable.

Definition 5.3.2 *(transition rule for* ask*)*
Let θ be a ground substitution such that $dom(\theta) = \mathsf{Free}(\psi)$.

$$\frac{\psi\theta \text{ is closed}}{\langle \mathsf{ask}(a, \psi), \sigma \rangle_V \xrightarrow{a?_i\psi\theta}_\theta \langle E, \sigma \rangle}$$

It is up to the receiving agent to compute an answer and - in case there exists more than one possible answer - to select an answer it is interested in. The computational burden thus is on the receiving agent which has to *deduce* the requested information by using the received information. The answer is computed by deducing which θ satisfy $\sigma \cup \varphi \models \psi\theta$.

Remark 5.3.3 It is possible to relax the requirement that θ grounds the question ψ and specifies a complete answer. In that case, *partial* answers might be provided by the sending agent. From a programming perspective, however, we believe that partial answers only complicate the programming task and such a feature can be simulated by existentially quantifying variables in a question.

The Communication of Information

Actual communication concerning the exchange of information between two agents occurs when both agents address each other and the sending agent provides the information from which the receiving agent is able to deduce an answer to its question. The transition rule for actual communication is defined next. The computed answer to the question of agent b is implicit in the transition associated with agent b (cf. the transition rule for ask).

Notation 5.3.4 If $\mathcal{M}$ is a set of agents and A is an agent, the notation $\mathcal{M}, A$ is used to denote the union of $\mathcal{M} \cup \{A\}$.

Definition 5.3.5 *(exchange of information)*
Let $A = \langle a, \Pi_a, \sigma_a, \Gamma_a \rangle$ and $B = \langle b, \Pi_b, \sigma_b, \Gamma_b \rangle$ be two agents such that $a \neq b$, and let $\mathcal{M}$ be a (possibly empty) multi-agent system such that $A \notin \mathcal{M}, B \notin \mathcal{M}$.

$$\frac{A \xrightarrow{b!_i\varphi} A' \ , \ B \xrightarrow{a?_i\psi} B' \text{ and } \sigma_b \cup \varphi \models \psi}{\mathcal{M}, A, B \longrightarrow \mathcal{M}, A', B'}$$

Example 5.3.6 In this example, we illustrate the semantics of definition 5.3.5. Consider two agents John and Roger which are going to a meeting to discuss a paper. A question concerning this meeting might be where the meeting takes place. Assume that Roger seeks to find an answer to this question and John is able and prepared to provide the information. The appropriate communicative action for Roger then would be the following:

$$\mathsf{ask}(\mathit{john}, \mathit{meet}(\mathit{paper}, 10:00\mathit{Fri}, 1:00, \mathit{Location}, \{\mathit{john}, \mathit{roger}, \mathit{mark}\}))$$

The free variable *Location* indicates that the question is about the location of the meeting. To provide an answer to Roger's question, John might send different messages depending on the beliefs of Roger. For example, John could simply tell Roger the place by:

$$\mathsf{tell}(\mathit{roger}, \mathit{meet}(\mathit{paper}, 10:00\mathit{Fri}, 1:00, \mathit{amsterdam}, \{\mathit{john}, \mathit{roger}, \mathit{mark}\}))$$

It is easy to see that Roger can deduce an answer from John's message. An alternative to sending this message would be to send a message from which Roger, given his current beliefs, could derive the location of the meeting. For example, assume that Roger takes the location of John at 10:00 on Friday to be definite information as to where the meeting takes place. Roger's beliefs would then imply the proposition

$$\begin{array}{l} \mathit{location}(\mathit{john}, 10:00\mathit{Fri}, \mathit{Loc}) \rightarrow \\ \quad \mathit{meet}(\mathit{paper}, 10:00\mathit{Fri}, 1:00, \mathit{Loc}, \{\mathit{john}, \mathit{roger}, \mathit{mark}\}) \end{array}$$

and John could send Roger a message concerning his location 10:00 on Friday from which Roger, given his beliefs, would be able to derive an answer to his question:

$$\mathsf{tell}(\mathit{roger}, \mathit{location}(\mathit{john}, 10:00\mathit{Fri}, \mathit{amsterdam}))$$

Alternative Views on the Exchange of Information

The idea to use deduction to formalise the exchange of information is captured in a simple and elegant way in definition 5.3.5. The particular implementation of this idea, however, is only one out of a range of alternative definitions. A number of interesting variations on a 'deductive' semantics are possible. As our main interest here concerns the receiver, we comment on one interesting alternative from the point of view of the receiving agent. In the transition rule for the exchange of information 5.3.5, the message φ is taken as additional information that - for the purpose of computing an answer - is added to the beliefs of the receiving agent. This type of belief change is also called *expansion* in the belief revision literature (Gärdenfors 1988). One consequence of this semantics is that if the message φ sent by agent a is inconsistent with the beliefs of agent b, any answer whatsoever is considered acceptable. Although we believe there is nothing wrong with this particular semantics, this observation suggests an interesting alternative semantics. The suggestion is to *update* - by using an update operator $\circ$ - instead of *expand* the beliefs of agent b with the message φ such that $\sigma_b \circ \varphi$ remains consistent. The informal interpretation of this type of information exchange should then be changed into: if agent b *would* believe φ to be true (after updating), the question ψ can be answered. There are still a number of other interesting alternatives to definition 5.3.5 for dealing with inconsistency. For example, a message inconsistent with the receiving agent's beliefs could simply be considered unacceptable.

Example 5.3.7 We now give an example which is more naturally handled by the alternative semantics in which the beliefs of the receiving agent are *updated* by the message that is received. Consider the situation in which Roger wants to know how long it would take Mark at 10:00 on Tuesday to get to Amsterdam, and currently believes (among other things) that Mark is in Utrecht. This is represented by $location(mark, 10:00\,Tue, utrecht)$:

$$\begin{array}{l}
location(mark, 10:00\,Tue, utrecht)\wedge \\
\forall P, T, L1, L2((location(P, T, L1)\wedge \\
\qquad location(P, T, L2)) \to L1 = L2)\wedge \\
transport(train, utrecht, amsterdam, 0:45)\wedge \\
transport(train, rotterdam, amsterdam, 1:00)\wedge \\
utrecht \neq rotterdam
\end{array}$$

But now suppose that Roger is not sure that Mark actually is in Utrecht and therefore asks Mark to provide the information:

$$\begin{array}{l}
\mathsf{ask}(mark, location(mark, 10:00\,Tue, FromLoc)\wedge \\
\qquad transport(train, FromLoc, amsterdam, TransTime))
\end{array}$$

Because Roger knows about transportation times, Mark just needs to inform Roger of his whereabouts from which Roger then would be able to deduce the time needed by Mark to travel to Amsterdam. However, if Mark would tell Roger:

$$\mathsf{tell}(roger, location(mark, 10:00\,Tue, rotterdam))$$

the semantics of definition 5.3.5 would render the belief base resulting from adding this information to Roger's beliefs inconsistent. The alternative semantics, where Roger's beliefs are updated with the message, however, would be able to deal with this type of situation and could provide a correct answer to Roger's question (depending on the semantics of the update operator $\circ$, of course).

Constraints on Mental States

Although our focus has been mainly on the receiving agent so far, there is no reason to suppose that we cannot impose constraints on the mental state of the sending agent. It is possible to *program* particular conditions on the mental state of the sender. For example, we can define a new communication primitive inform by using the primitive tell that tests whether or not the sending agent believes what it tells:

$$\mathsf{inform}(a, \varphi) \overset{df}{=} \varphi?;\ \mathsf{tell}(a, \varphi)$$

The inform primitive thus requires the sender to be *honest*. Because we can program such conditions, we did not incorporate these conditions in the semantics of tell. Another reason why we did not include such conditions is that by not doing so we have been able to keep the semantics of our primitives basic and simple. This makes it easier to understand the meaning of the primitives and to use them to program agents. A somewhat more philosophical argument is that from the perspective of speech act theory one could argue that sincerity conditions on the mental state of the agent should not be part of the (semantic) definition of speech acts (cf. Bach & Harnish (1979)). The point is that if agent a is lying to agent b that φ is the case, agent a can still be described as informing or telling agent b that φ.

Although the inform action first performs a check on the belief base to see whether or not the message φ is believed, there is no guarantee that the agent still believes φ at the moment of sending the message. The reason is that an agent is multi-threaded and the defined action inform is not atomic. As a consequence, some other goal of the agent may have interfered and caused a change in the belief base of the agent *after* performing φ? but before communication by means of tell has taken place. However, if no other goal of the agent interrupts the execution of the defined primitive inform by changing the belief of the agent that φ is the case, then the primitive inform succeeds only if the agent believes what it tells.

Also note that we do not require that the receiving agent adds the information it receives to its belief base. The successful processing of a message in our view only consists of the computation of an answer that is deduced from the message. This does not mean that the receiving agent cannot update its mental state with new information. Of course, the programming language offers other facilities for updating the beliefs of the receiving agent. For example, it is not difficult to program an agent that after performing an ask action updates its mental state with the answer that it received. Such an agent should simply insert the information in its belief base:

$$\mathsf{ask}(a, \psi);\ \mathsf{ins}(\psi)$$

Note that this solution only works for answers that are consistent with the current belief base of the agent.

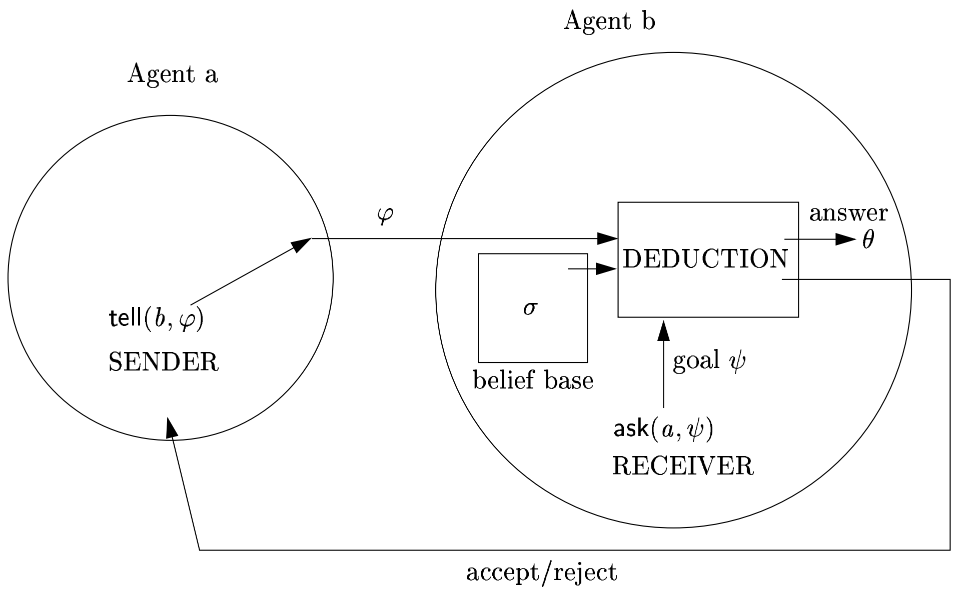

Figure 5.1: Information exchange by means of deduction

In figure 5.1, the communication of information between two agents is illustrated. The sending agent a, which is the agent using the communicative action $\mathsf{tell}(b, \varphi)$, communicates to agent b the proposition φ. When this proposition is received by agent b and that agent has a matching $\mathsf{ask}(a, \psi)$ indicating that it is willing to communicate with a, agent b uses the received information to compute an answer to its question. The received proposition φ and the current belief base of agent b are inputs for a deductive system, like for example Prolog, and the deductive system is given the question ψ as a *goal*, which means it should attempt to compute an *answer* for ψ. In case the deductive system succeeds, it outputs an answer θ. The answer is used in further computations of agent b. In case the deductive system indicates whether it succeeded or failed, this information is (implicitly) communicated to agent a (an accept is sent in case of success, a reject in case of a failure).

5.4 Requests

The second pair of communication primitives we introduce may be used to exchange requests. Again, we are interested in capturing what could be considered as the successful processing of a request by the receiver. The main issue for the receiving agent, we believe, is whether or not the agent is able to compute a goal or plan which would satisfy the request. This is the first issue an agent which receives a request should deal with and is a prerequisite for the evaluation of a request. Therefore, we take the successful processing of a request to mean that the receiving agent is able to make a proposal that would satisfy the request.

Now consider the requirements on such a proposal. First of all, we should make clear that we consider declarative proposals and declarative requests here. That is, requests and proposals are propositions or statements from the knowledge representation language $\mathcal{L}$. Given a request φ which specifies a state of affairs to be achieved, for a proposal ψ to satisfy the request, this proposal should entail that a state of affairs such that φ is obtained (given what the agent which makes the proposal currently believes). Now we notice that the problem of finding a proposal ψ such that $\sigma \cup \psi \models \varphi$, that is, a proposal ψ which given the current beliefs σ entails φ, is analogous to an *abductive problem.* This suggests that an appropriate semantics which captures the successful processing of requests by the receiver can be obtained by using an *abductive semantics.*

Abduction is sometimes paraphrased as reasoning from an observation to an explanation, or alternatively, from an effect to a cause. In a communication setting, where more than one agent is involved, we think of the agent sending a request as the agent which desires a particular effect to be realised, and of the agent who receives the request as the agent which tries to compute a cause that would establish the effect. The basic idea underlying the semantics of the primitives is that abductive reasoning can be used by the receiving agent to compute a proposal that *would* achieve the request of the sender.

Perhaps even more than in the case of the exchange of information we should emphasise that the semantics for req and offer that we propose here is not intended to capture the much more complicated semantics of the speech act of requesting. We are interested in the way the receiving agent processes a request that it has received and in providing an agent with the computational means to do so. We suggest that an abductive semantics provides an appropriate model for the processing of a request.

Before we proceed, we give a brief, somewhat more formal, outline of an *abductive problem.* An abductive problem can be characterised as follows: Given a *background theory* T, a set of *hypotheses* H, and an *effect* φ, the associated *abductive problem* is to find an (instance of a) hypothesis $h \in H$ such that $T \cup h \models \varphi$; moreover, $T \cup h$ is required to be consistent (cf. Poole (1989); this is the strong notion of abduction). A hypothesis h is generally considered to be a solution to an abductive problem only if a number of additional requirements are satisfied (cf. Mayer & Pirri (1996)). For example, a solution to an abductive problem should be as simple as possible. More general, some hypotheses are *preferred* over others. Another requirement on a solution is that it should be a most specific instance of a hypothesis. A solution h is also said to *cover* φ.

Making a Request

The communicative action $\mathsf{req}(b, \varphi)$ is an action to send a message φ to agent b. There are *no* constraints on the sending agent imposed by the semantics; the only constraint is that agent b, to whom the message is sent, is able to offer a proposal which would establish φ. The informal interpretation we associate with the action $\mathsf{req}(b, \varphi)$ is that φ expresses a *request.*

The request φ is required to be definite, i.e. we do not allow the occurrence of free variables in φ *when* $\mathsf{req}(b, \varphi)$ *is executed.* So, any free variables which occur in a $\mathsf{req}(b, \varphi)$ in a plan of an agent must have been instantiated by retrieving bindings for these variables and parameter passing before the request φ can be communicated. The agent which makes the request is not automatically informed of the type of proposal the receiving agent b makes. The sender does not receive any information, which explains the empty substitution $\varnothing$ in the transition rule.

The symbol $!_r$ in the label associated with the transition below is used to distinguish between the two types of message exchange, the exchange of a request and the exchange of information. As before the exclamation mark ! indicates that a message is being sent, while the subscript r indicates the type of communication, namely the exchange of a request. A similar remark as in the case of the transition rules for tell and ask applies here. That is, the transition rules for req and offer define *virtual* computation steps or attempts.

Definition 5.4.1 *(transition rule for* req*)*

$$\frac{\varphi \text{ closed}}{\langle \mathsf{req}(b, \varphi), \sigma \rangle_V \xrightarrow{a!_r\varphi}_{\varnothing} \langle E, \sigma \rangle}$$

Offering a Proposal

The communicative action $\mathsf{offer}(a, \psi)$ is an action which specifies a *proposal* ψ to achieve a request of agent a. Informally, the action $\mathsf{offer}(a, \psi)$ could be viewed as advertising that the agent is prepared to offer ψ to agent a; alternatively, the agent offers a *proposal* ψ as a way to achieve a request of agent a.

The proposal ψ may contain free variables which indicate the *range* of proposals the agent is prepared to make. That is, by offering ψ any one of the instances of the free variables in ψ is offered. The receiving agent needs to process a received request and compute a specific instance of its proposal ψ which would satisfy the request. The process involved, as we suggested, is that of abduction. The receiving agent attempts to *abduce* suitable instances θ for the free variables in ψ such that, given the agent's current beliefs σ, the request φ of agent a is entailed by the proposal, i.e. $\sigma \cup \psi\theta \models \varphi$; moreover, the computed proposal $\psi\theta$ must be consistent with the agent's beliefs σ, i.e. $\sigma \not\models \neg\psi\theta$.

In the terminology of an abductive problem, the belief base of the receiving agent corresponds to the background theory; the request φ corresponds to the effect; and ψ is the hypothesis which the receiving agent uses to find a solution to the abductive problem. There may be more than one instantiation of the free variables in the proposal ψ which would satisfy the request of agent a. In that case, the receiving agent is free to choose one according to its own preferences.

The requirement that $\psi\theta$ is closed in the transition rule below is explained by the fact that we require an agent to offer a most specific proposal.

In the transition rule for offer, the question mark ? in the label indicates that the offering agent is the receiving agent. The subscript r indicates that the communication involves the exchange of a request. The fact that there are no restrictions imposed on the substitution θ in the transition rule indicates that the receiving agent in principle is prepared to offer any instance of ψ as a proposal to achieve a request.

Definition 5.4.2 *(transition rule for* offer*)*
Let θ be a substitution such that $dom(\theta) = \mathsf{Free}(\psi)$.

$$\frac{\psi\theta \text{ is closed}}{\langle \mathsf{offer}(a, \psi), \sigma\rangle_V \xrightarrow{a?_r\psi\theta}_\theta \langle E, \sigma\rangle}$$

Remark 5.4.3 By using the $\mathsf{offer}(a, \psi)$ primitive, the set of hypotheses associated with the abductive problem in the semantics of requests is a singleton $\{\psi\}$. It is, however, possible for communicating agents to use abductive problems in their full generality. That is, it is possible to implement an action $\mathsf{offer}(a, H)$ where H is a set of hypotheses, and the associated abductive problem is to find an instance of one hypothesis from H that covers the request made by another agent. This more general communicative action can be implemented by

$$\mathsf{offer}(a, \{h_1, \ldots, h_n\}) \overset{df}{=} \mathsf{offer}(a, h_1) + \ldots + \mathsf{offer}(a, h_n)$$

The Communication of a Request
Actual communication which involves the exchange of a request between two agents occurs when both agents address each other and the receiving agent is able to abduce a specific proposal which would satisfy the request φ of the sending agent; the receiving agent, moreover, only offers a proposal which is consistent with what it believes to be true. The latter condition prevents the receiving agent from offering too strong proposals like $\perp$ (false) which would satisfy any request. The transition rule for the exchange of a request is defined next. The computed substitution θ to instantiate the proposal is implicit in the transition associated with agent b (cf. the transition rule for offer).

Definition 5.4.4 *(exchange of a requests)*
Let $A = \langle a, \Pi_a, \sigma_a, \Gamma_a\rangle$ and $B = \langle b, \Pi_b, \sigma_b, \Gamma_b\rangle$ be two agents such that $a \neq b$, and let $\mathcal{M}$ be a (possibly empty) multi-agent system such that $A \notin \mathcal{M}, B \notin \mathcal{M}$.

$$\frac{A \xrightarrow{b!_r\varphi} A' \, , \, B \xrightarrow{a?_r\psi} B' \, , \sigma_b \not\models \neg\psi, \sigma_b \cup \psi \models \varphi}{\mathcal{M}, A, B \longrightarrow \mathcal{M}, A', B'}$$

Example 5.4.5 In this example, we illustrate the semantics of definition 5.4.4. Consider again the meeting example that involved agents John and Roger, but now suppose that agent John would like to meet to discuss a paper with agent Roger 10:00Fri. Then agent John could use the communicative action req to communicate his request to meet to agent Roger as follows:

$$\mathsf{req}(roger, meet(paper, 10:00Fri, 1:00, amsterdam, \{roger, john\}))$$

Also assume that agent Roger is prepared to make the following proposal to achieve a request of John by means of the communicative action offer:

$$\mathsf{offer}(john, meet(paper, 10:00Fri, 1:00, AnyPlace, \{roger, john\}))$$

Here, the free variable *AnyPlace* indicates that Roger is prepared to offer a proposal to meet John at *any place* 10:00 on Friday in reply to a request of John. We want to emphasise that all agent Roger does by means of the action offer is to try and find a proposal which would satisfy a request of John. There are no conditions on the mental state of the speaker similar to those usually associated with the speech act of requesting, and the actions req and offer are not intended as the equivalents of such a speech act. The action offer provides the receiving agent with the computational means to process a request of another agent in the sense outlined above. In the example, agent Roger uses abduction to compute a specific instance of his proposal that would satisfy the request of John. As is clear in this simple case, the only instance (given that Roger does not believe anything about meetings 10:00 on Friday) which would satisfy the request is the proposal to meet in Amsterdam.

The example illustrates the use of variables in the proposal of the offering agent. A variable in a proposal ψ means that the proposal is a proposal to satisfy *any specific request* of the requesting agent concerning this parameter in the proposal, as long as this is consistent with the beliefs of the agent. In the example, the agent offers a proposal to meet at any requested place. It does not mean, however, that the agent offers *every* instance of its proposal. That is, it does not offer the universal closure $\forall\psi$ as a way to achieve a request. And neither would the requesting agent be satisfied by such a proposal in the meeting example (it does not even make sense to meet at *all possible* places at a particular time). We assume that the requesting agent would like the other agent to satisfy its request by making a proposal that is as concrete as possible. This explains why the agent that is computing a proposal abduces a *most specific instance* of its proposal which would satisfy the request.

The distinguishing feature of the semantics of req and offer is the consistency constraint imposed on any proposals offered to satisfy a request. This constraint can be used to enforce that certain preferences of the receiving agent which computes a proposal are taken into account. The consistency constraint is also used for this purpose in the implementation of the negotiation protocol in the next chapter.

Example 5.4.6 Consider again the example of agent John requesting a meeting with agent Roger. However, in this case, agent John is requesting to meet with agent Roger somewhere, and does not care for any particular place. Agent John could communicate this request by means of the following action:

$$\mathsf{req}(roger, \exists L(meet(paper, 10:00Fri, 1:00, L, \{roger, john\})))$$

Again, assume that agent Roger is prepared to make the same proposal as before:

$$\mathsf{offer}(roger, meet(paper, 10:00Fri, 1:00, AnyPlace, \{roger, john\}))$$

But in this case, we also assume that agent Roger has a preference to meet in either Utrecht or Rotterdam, and is not prepared to meet in any other place. This type of preference can be coded into the beliefs of the agent. This is achieved by adding a constraint in the belief base of agent Roger as follows:

$$\begin{array}{l}\forall\, Id, T, Len, L, P(meet(Id, T, Len, L, P) \\ \qquad\qquad \rightarrow (L = utrecht \vee L = rotterdam))\end{array}$$

Given this preference, the consistency constraint in the semantics makes sure that agent Roger only abduces proposals for meeting in Utrecht or Rotterdam because any instance of his proposal

$$meet(paper, 10:00Fri, 1:00, AnyPlace, \{roger, john\})$$

must be consistent with his beliefs. A disadvantage, however, of this way of representing preferences is that if agent John would have requested to meet in Amsterdam, agent Roger would not have been able to compute a proposal which would satisfy this request.

Alternative Views on the Exchange of Requests

The semantics for the exchange of a request in definition 5.4.4 is a simple instance of an abductive semantics for this type of communication. Again, there are a number of interesting variations on such a semantics. A particularly interesting alternative which already is suggested by the use of abduction is to extend the semantics with a *preference relation* on the proposals an agent is prepared to make. In this context, however, the preference relation would not primarily be based on properties of simplicity of the proposal as is usual in the literature on abduction, but would be based on concepts associated with the interests of agents, like efficiency, for example. One reason why we did not incorporate this extension in the semantics is that we want to study the basic ideas in their most simple form first. Also, as illustrated in example 5.4.6, the consistency constraint in definition 5.4.4 can be used to deal with preferences of an agent. There are various other ways of modifying the semantics of definition 5.4.4. For example, in the case of exchanging requests we could also argue that a proposal should not simply be added to the current beliefs as is done in definition 5.4.4, but should be used to *update* the current beliefs and to check that if the agent *would* believe that its proposal were true, the agent also would believe that the request is satisfied.

Example 5.4.7 We give an example which illustrates the use of the alternative semantics that incorporates a preference relation on proposals. Again, we use the example concerning the request to meet somewhere. We assume that a preference order can be represented as follows:

$$\begin{array}{l} meet(Id,\, T, L, utrecht, Att) > \\ meet(Id,\, T, L, rotterdam, Att) > \\ meet(Id,\, T, L, amsterdam, Att) \end{array}$$

Informally, this means that for any particular instance of the variables Id, T, L and Att, a meeting in Utrecht is preferred over a meeting in Rotterdam, which in turn is preferred over a meeting in Amsterdam. Now assume that the same request is made by Roger

$$\mathsf{req}(john, \exists\, L(meet(paper, 10:00Fri, 1:00, L, \{roger, john\})))$$

and the same proposal is offered by John:

$$\mathsf{offer}(roger, meet(paper, 10:00Fri, 1:00, AnyPlace, \{roger, john\}))$$

In this case, Roger would again compute a proposal to meet in Utrecht because this is the place he most prefers to meet. However, in case John would have requested to meet in Amsterdam, Roger would also have been able to come up with a proposal which would satisfy this request, even though Amsterdam is ranked lowest in his preference order. Of course, a proposal can only be found in case we assume that Roger has no conflicting beliefs that are inconsistent with that proposal.

Acting on a Request

Because our main objective is to provide the receiving agent with some appropriate computational means to process a request, in the formal semantics no requirements on the mental state of the requesting agent are formulated. Moreover, the agent communicating a request to some other agent is not informed by that agent what proposal was computed. And neither does the requesting agent receive any information whether or not its request will be achieved by the other agent; this depends on the particular goals and plans of the agent receiving the request. The semantics of the exchange of a request also does not specify that the agent which receives the request is *capable* of achieving the proposal which is computed in reply to a request. This is analogous to the fact that there is no honesty condition associated with the tell primitive. However, again the programming language offers the facilities to actually achieve requests, and it is up to the programmer to *program* a particular plan to deal with requests. For example, agent Roger may update the agenda in its belief base accordingly after computing a proposal upon receiving a request of John. The following plan would do. Note that the offer action computes a binding for the free variable *AnyPlace* which is passed onto the ins action.

$$\begin{array}{l} \mathsf{offer}(john, meet(paper, 10:00Fri, 1:00, AnyPlace, \{john, roger\})); \\ \mathsf{ins}(meet(paper, 10:00Fri, 1:00, AnyPlace, \{john, roger\})) \end{array}$$

It is important to realise here that in the examples that were presented the agents are personal assistants that maintain the agendas of their users and support meeting scheduling. The task of such an agent is to keep the agenda of its user up to date. The items stored in the agenda represent commitments of the user to meetings. Therefore, an update of the belief base actually represents a new commitment of the user to go to a meeting. Of course, it is up to the user to act upon his commitment, and no such thing can be expected from the software agent itself.

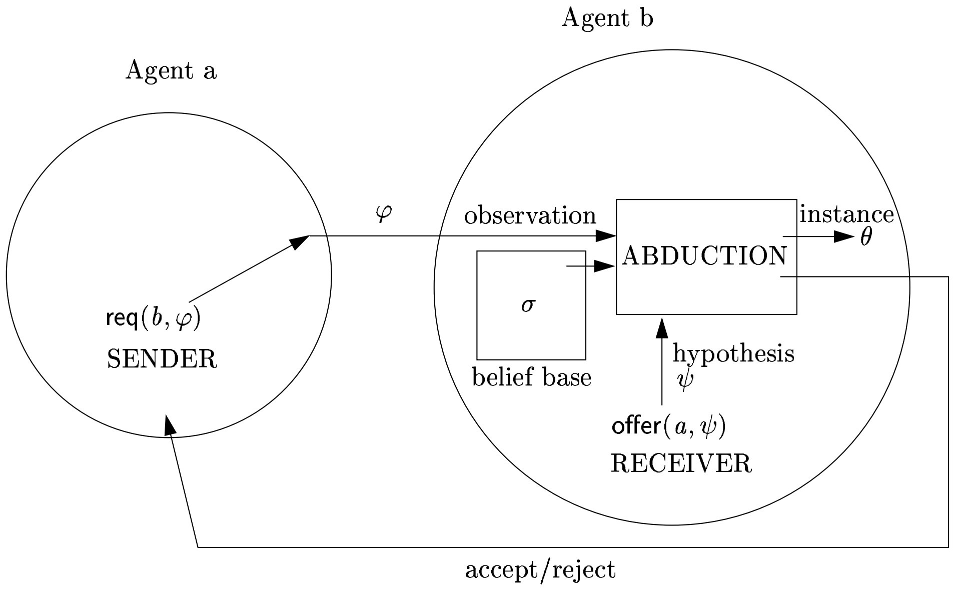

Figure 5.2: Requesting and offering by means of abduction

In figure 5.2, the requesting and offering between two agents is illustrated. The sending agent a, which is the agent using the communicative action $\mathsf{req}(b, \varphi)$, communicates to agent b the proposition φ. When this proposition is received by agent b and that agent has a matching $\mathsf{offer}(a, \psi)$ indicating that it is willing to communicate with a, agent b uses the received information to compute a most specific *instance* of its offer. Recall that an abductive problem consisted of a background theory T, a hypothesis h and observation o. To compute the instance, agent b uses an *abductive system*. The received proposition φ is inputted as the *observation* into this system. The current belief base of agent b is inputted as the *background theory*. And the abductive system is asked to compute an instance of *hypothesis* ψ. In case the abductive system succeeds, it outputs an instance θ. The instance θ is used in further computations of agent b. In case the abductive system indicates whether it succeeded or failed, this information is also communicated to agent a (accept in case of success, reject in case of a failure).

Requests for Explanations It is possible to use the req and offer primitives for asking *why*-questions. Such questions may be characterised as requests for an explanation. An abductive system can be used to answer such questions and therefore the offer primitive can be used to provide answers to *why*-questions.

Example 5.4.8 Consider agent Roger again, who this time is wondering why John will go to Amsterdam next Friday. Since scheduled activities are reasons for agents to be at certain locations, the request can be formulated by:

$$\mathsf{req}(john, meet(\{course, paper, workshop\}, 11:00Fri, amsterdam, \{john\})$$

Note that we introduced a new notation for formulating requests. The expression $\{course, paper, workshop\}$ is a notational device denoting the possible explanations agent John allows or thinks are possible. Such a request is *not* equivalent to a disjunctive request made up of the three disjuncts corresponding to the alternatives. A disjunctive request might trivially be explained by the disjunction itself, but the request of John is a request to explain *one* of the alternatives indicated. It is equivalent, however, to the following nondeterministic goal which is a request to either explain that John has a course to teach, a paper to discuss, or a workshop.

$$\begin{aligned}&\mathsf{req}(john, meet(course, 11:00Fri, amsterdam, \{john\}) +\\&\mathsf{req}(john, agenda(paper, 11:00Fri, amsterdam, \{john\}) +\\&\mathsf{req}(john, agenda(workshop, 11:00Fri, amsterdam, \{john\})\end{aligned}$$

Agent John might simply offer one of the three alternatives as an explanation. For example, agent John could explain that it will go to Amsterdam because of a course it has to teach there:

$$\mathsf{offer}(roger, meet(course, 11:00Fri, amsterdam, \{john\}))$$

The explanation, however, still needs to be told to Roger. The receiving agent is only induced to explain the observation of the other agent. The agent could communicate how it explains the observation by means of tell.

Notice that the use of nondeterministic choice to implement the notational device introduced in this example is similar to the use of nondeterministic choice to generalise an offer to a set of hypotheses in remark 5.4.3.

5.5 Approaches Based on Speech Act Theory

The main approaches in the literature involving agent communication languages like KQML (Labrou & Finin 1994) or FIPA-ACL (FIPA 1998) are based on speech act theory (Searle 1969). Before we discuss these other approaches, we therefore first provide a short summary to introduce the most important concepts of speech act theory. One of the basic insights of speech act theory is that the conveying of a message not only consists in making a statement, but in a broader sense constitutes a communicative action: making a promise, begging, informing, etc. are all different kinds of communicative actions.

In speech act theory, communicative actions are decomposed into locutionary, illocutionary and perlocutionary acts. A locutionary act refers to what the speaker says; the illocutionary act refers to what the speaker does by saying such-and-such; and the perlocutionary act refers to how the speaker affects the hearer by saying such-and-such. For example, a speaker may say that it is six o'clock (locutionary act); by saying so the speaker may be requesting the hearer to leave the shop (illocutionary act); and the effect on the hearer may be that (s)he actually leaves the shop (perlocutionary act).

By performing an illocutionary act, a speaker expresses mental attitudes: beliefs and desires. Accordingly, one can classify types of illocutionary acts in terms of expressed attitudes. Many classifications of speech acts have been proposed. The one we summarise here is based on that of Bach and Harnish (Bach & Harnish 1979). They distinguish four main classes of speech acts: (i) *constatives*, like assertives and predictives, which express belief, (ii) *directives*, like requestives and questions, which express a wish concerning some prospective action by the hearer, (iii) *commissives*, like promises and offers, which express an intention and belief that the speech act obligates the speaker to do something, and (iv) *acknowledgements*, like apologise and greet, which express feelings regarding the hearer.

Note that although a speech act is a means of expressing attitudes, the speaker does not actually have to have the expressed attitudes. As Bach and Harnish (Bach & Harnish 1979), p. 39 write:

> To *express* an attitude in uttering something is [...] to (..) intend that the hearer take one's utterance as a reason to believe one has the attitude. The speaker need not have the attitude expressed, and the hearer need not form a corresponding attitude. The speaker's having the attitude expressed is the mark of sincerity. If the hearer forms a corresponding attitude that the speaker intended him to form, the speaker has achieved a perlocutionary effect in addition to illocutionary uptake.

As we discussed, our communication primitives are not intended to model speech acts. Nevertheless, in a loose sense, we can classify our primitives as belonging to one of the classes outlined above. The tell primitive, for example, can be classified as a kind of constative. The req and ask can be classified as directives. And the offer can be classified as a commissive. The communication primitives, however, should not be taken to correspond to any speech act in particular. The tell primitive, for example, does not correspond to informing only, or suggesting only. The communication primitives can be used for many different purposes. The tell can be used both to assert and to predict; the offer primitive can both be used to promise and to make an offer. So, the informal meaning associated with the names of the communication primitives should not be confused with the formal semantics based on deduction and abduction.

In our semantics for the communication primitives we did not include any sincerity conditions on the actual mental state of the speaker. For example, it is not required that the agent using tell actually believes what it tells. Also, no *perlocutionary effects* are incorporated in the semantics of the primitives. The way an agent deals with a particular message of another agent is not specified in the semantics of the communication primitives (and is not part of the speech act per se, cf. the quote from Bach & Harnish (1979) above). For example, the agent receiving a proposition communicated by tell does not automatically update its belief base with this proposition.

Our approach is based on the view that multi-agent level communication should be based on a well-defined and clear semantics and at the same time should be kept as simple as possible. Moreover, our focus has been more on the hearer than on the speaker.

We take a somewhat different perspective as to how speech act theory could be of use in the design and development of agents. Our view is that speech act theory may provide a useful set of *abstract descriptions* of communicative actions for agents which can be used to *specify* and *verify* the behaviour of agents. Speech acts thus would play a role in the *proof theory* associated with the agent programming language. In the associated programming logic, the speech acts could be derived just like, for example, the mental attitude of belief would be derived from the program text and its semantics. Instead of *defining* a set of communication primitives corresponding to all kinds of subtle distinctions related to speech acts, we propose to *derive* what kind of speech act has been performed by an agent from the characteristics of the content proposition and the mental state of the agent. Alternatively, a formal specification in the logic may specify that an agent should perform a type of speech act. In that case it is up to the programmer to satisfy the specification.[1] The arguments are similar to, but not the same, as some of those put forward in (Cohen & Levesque 1990*b*) (cf. also Colombetti (2000)). In (Cohen & Levesque 1990*b*) a *logical* approach to modelling speech acts is discussed. In our case, we argue that a distinction should be made between actions provided for in a *programming language* and actions which are defined in a *logic*. Moreover, we do introduce a number of communication primitives in contrast with the approach in (Cohen & Levesque 1990*b*). We think speech act theory should not be viewed as a repository of all kinds of different communicative acts for which computational equivalents should be provided in an agent communication language, as is done in KQML or FIPA-ACL.

The view outlined has a number of advantages. First of all, the agent communication language is kept simple. This prevents all kinds of confusions on the part of the programmer as to what kind of communication primitive it should use

[1]For a similar remark cf. Labrou & Finin (1994), where it is suggested that the propositional content conditions should be taken care of by the programmer; and indeed, it is hard to see how these conditions could be guaranteed in any other way in KQML-like languages. Cf. also the discussion later in this section.

in a particular case. Second, the idea to derive speech acts from the semantics of the agent (communication) language seems to provide for a more principled approach than the approach which is used to provide semantics for KQML or FIPA-ACL. It prevents unnecessary complications in the semantics and prevents the tendency to incorporate 'everything' into the semantics of speech acts. The different concerns which arise in the context of communicating agents are not clearly separated in the approaches mentioned, we think (although this is more true of KQML than of FIPA-ACL).

For example, in (Labrou & Finin 1994) an attempt is made to incorporate various aspects of illocutionary acts *and* conversation policies into the semantics of speech acts. There are several disadvantages with such an approach. On the one hand, including conversation policies in the semantics provides a burden on the programmer, who has to deal with these policies. For example, we think that a requirement to reply in one of several ways to a communicative action of another agent should not be part of the semantics of a speech act. On the other hand, there is a methodological reason. If conversation policies are integrated into the semantics of speech acts, it is difficult to see where to draw the line. For example, is the Contract Net Protocol a conversation policy or is it a protocol which is not part of the semantics of speech acts? In our view, it is better to separate these issues.

Moreover, some of the speech acts provided in KQML seem to have arisen more from a computational concern than from a clear and formal perspective on speech acts. For example, the performative **stream-all** is used to ask an agent to provide all known answers to a question *in a particular format*. Another performative **ask-all** is introduced which provides for yet another format of response. A similar point can be made regarding the idea to introduce specific communication primitives to communicate with a so-called facilitator. A facilitator is a special agent which is introduced to manage particular aspects of the communication between agents, and a subset of the communication primitives supplied in KQML is used only to deal with this facilitator. In our view, these features should not be included in the semantics of an agent communication language.

Finally, the specification of the semantics of the communication primitives in KQML and FIPA in a modal logic including operators for belief, intention, uncertainty, etc. poses a problem for integrating these communication languages into agent frameworks, i.e. agent programming languages or architectures. The problem is that the semantics is quite abstract and it is hard to see how to *ground* the modal semantics in an arbitrary agent framework. To put it in another way, how do we know that an agent in a particular agent framework actually does have the required mental attitudes, etc. corresponding to the modal semantics of the operators? It has not yet been shown how to demonstrate a link between a formal modal logic and an operational agent framework. The problem is sometimes referred to as the *gap between theory and practice*. In part III, we will discuss a solution to this problem. Note however, that our framework does not suffer from these problems. The communication primitives presented in this chapter naturally fit into the agent language 3APL.

5.6 Conclusion

This chapter introduced communication primitives for 3APL agents. Two sets of primitives were defined. One set particularly designed for the exchange of information, and a second set designed for the communication of requests. The operational semantics associated with these primitives focuses on the processing of a received message by the hearer. The main concern of the hearer in case of information exchange is identified with the *computation of an answer to a question.* The process associated with this type of computation is the process of *logical deduction.* The semantics of the communication primitives for information exchange - tell and ask - is therefore based on deductive reasoning. The main concern of the hearer in case of the exchange of a request is identified with the *computation of a proposal that would satisfy the request.* The process associated with this type of computation is the process of *abductive reasoning.* The semantics of the communication primitives for the exchange of requests - req and offer - is therefore based on abductive reasoning. Both the reasoning processes of deduction and abduction are well-understood, and provide a clear and expressive semantics for communication between agents. There are some relations with work of Eijk et al. (1999) who introduces a similar semantics for the exchange of information. The use of an abductive semantics is also proposed in (Dragoni et al. to appear) for an inform primitive. The ideas behind the of work of Dragoni et al. are similar in a number of respects to that of the model for identifying a communicative act as a particular speech act as proposed in (Bach & Harnish 1979).

The two sets of communication primitives are integrated into the agent programming language 3APL. The semantics of the primitives is designed to fit in with the other primitives of the programming language. In this respect, our approach differs from other approaches that aim at designing a 'universal' communication language for agents like KQML and FIPA-ACL. Approaches that separate the communication features from other agent features, however, suffer from the problem that it is not clear what type of semantics would be appropriate for such communication languages. A second problem is that it is not clear how to integrate these communication languages into existing agent frameworks.

We argued that one should not aim at providing a (large) set of computational equivalents of speech acts. Instead, we believe, it is important to design a semantics for communication primitives that is both simple and intuitive. Our approach does not exclude the use of speech act theory, but we view speech act theory more as a tool for the specification and verification of communication between agents than as a framework for implementing inter-agent communication. Using speech act theory as an implementation framework is unwieldy and requires a programmer to distinguish too many subtle differences between communication primitives to be useful for programming agents.

CHAPTER 6

Meeting Scheduling

One of the more interesting applications of intelligent agents concerns their use as personal assistants for meeting scheduling. As we discussed in chapter 2, personal assistants deal with routine tasks and the associated personal preferences of their users for carrying out these tasks. Because the scheduling of meetings can be automated, and usually involves specific user preferences, the use of intelligent agents for the meeting scheduling domain is particularly well-suited.

The meeting scheduling domain involves multiple agents negotiating on an appropriate time for a meeting. To resolve any possible conflicts between these agents, they need to communicate about the meeting times that are acceptable. The meeting scheduling domain also allows us to discuss and illustrate the use of the communication primitives introduced in the previous chapter. In particular, we will discuss in more detail the differences between the primitives for information exchange and the primitives for the exchange of requests.

Although meeting scheduling is a complex problem, some solutions have been proposed in the literature (cf. Sen & Durfee (1996)). One of the solutions proposed, is to implement distributed meeting scheduling by using the *multi-stage negotiation protocol* (Conry et al. 1988). We show how to implement the multi-stage negotiation protocol in 3APL. First, we design a set of plans for two-agent meeting scheduling where only two agents need to agree on a meeting time. Then we generalise this solution to multi-agent meeting scheduling for an arbitrary number of agents.

6.1 Constraints on Meetings

The scheduling of meetings not only involves preferences of users, but also involves a number of constraints on meetings to ensure that the set of scheduled meetings is coherent. These coherence or integrity constraints concern, for example, the identification (by means of identifiers) and overlapping of meetings. Because it is hard to make sense of scheduled meetings without such constraints, we assume that the constraints enumerated in this section are incorporated as background knowledge on meetings in the belief bases of all agents. Of course, each agent may have more specific constraints that are associated with meetings - in particular constraints coding the preferences of a user - as long as they are consistent with the four integrity constraints listed below.

The first constraint requires that each scheduled meeting is assigned a unique meeting identifier. A meeting identifier thus uniquely corresponds with a scheduled meeting. The second constraint requires the sets of attendants of two different but overlapping meetings to be empty. The set of attendants of one meeting and that of another overlapping meeting are thus required to be disjoint. The third constraint imposes a minimum and a maximum length on a meeting. The constraint excludes the extreme case where a meeting does not take any time. The constraint on the length of meetings is a simple example of a constraint on meetings and, of course, different constraints concerning the length of a meeting cay be specified in different contexts. Finally, the fourth constraint is introduced to illustrate other possible constraints or preferences on meetings. It expresses that a meeting never should take place in the evening or during the night. In this chapter, for simplicity and notational convenience, we abstract from the locations of meetings.

Constraint 1: Meeting Identifiers refer to unique meetings:

$$\begin{aligned}&(\mathit{meet}(\mathit{MeetId}, T_1, L_1, \mathit{Att}1) \wedge \mathit{meet}(\mathit{MeetId}, T_2, L_2, \mathit{Att}2))\\ &\qquad \rightarrow (T_1 = T_2 \wedge L_1 = L_2 \wedge \mathit{Att}1 = \mathit{Att}2)\end{aligned}$$

Constraint 2: The attendants of two overlapping meetings are disjoint sets of people:

$$\begin{aligned}&(\mathit{meet}(\mathit{MeetId}1, T_1, L_1, \mathit{Att}1) \wedge \mathit{meet}(\mathit{MeetId}2, T_2, L_2, \mathit{Att}2) \wedge\\ &T_1 \leq T_2 < T_1 + L_1 \wedge \mathit{MeetId}1 \neq \mathit{MeetId}2)\\ &\qquad \rightarrow \mathit{Att}1 \cap \mathit{Att}2 = \varnothing\end{aligned}$$

Constraint 3: A meeting takes longer than 0:00 but less than 8:00 hours:

$$\mathit{meet}(\mathit{MeetId}, T, L, \mathit{Att}) \rightarrow 0:00 < L < 8:00$$

Constraint 4: There are no meetings scheduled between 18:00 and 8:00:

$$\mathit{meet}(\mathit{MeetId}, T, L, \mathit{Att}) \rightarrow (T \geq 8:00 \wedge T + L \leq 18:00)$$

6.2 The Multi-Stage Negotiation Protocol

The multi-stage negotiation protocol is used in our implementation of a multi-agent system for meeting scheduling. The multi-stage negotiation protocol allows agents to negotiate for several rounds to reach an agreement. The basic idea of the protocol is to designate one of the agents as the *host agent*. The other agents involved in the negotiation are called the *invitees.* The host agent initiates the negotiation and will try to arrange a meeting with the *invitees.*

Every stage of the negotiation protocol consists of two phases. The first phase is the *negotiation phase* in which proposals and counterproposals are exchanged. In the negotiation phase, the host starts by proposing a meeting time to all the invitees for the meeting. The invitees reply by either indicating *acceptance* of the meeting proposal or by *counterproposing* another meeting time. The second phase is called the *evaluation phase.* In this phase, the *host* evaluates the counterproposals and selects one of the proposals as the best proposal.

A particularly important property of negotiation protocols is whether or not they *converge.* The multi-stage negotiation protocol does not guarantee convergence without some extra assumptions. To guarantee that the negotiation will terminate with an appropriate solution, we therefore make a number of additional assumptions. First of all, we assume that all attendants of a meeting including the host and the invitees always share a free slot of appropriate length in their agenda such that the meeting can be scheduled at this slot. Secondly, the strategy of the participants during the negotiation will be to request to schedule the meeting at the *earliest meeting time* that is consistent with all the constraints and preferences of the attendants. In case everyone complies with this request, the host can compute an earliest possible meeting time each round of the negotiation. Finally, we assume that all scheduled meetings can be identified by unique identifiers and that a new label which is different from all labels used to refer to current meetings in the multi-agent system is associated with the meeting that is being negotiated.

6.3 Two-agent Meeting Scheduling

We first design a negotiation protocol for negotiation between two agents. In two-agent meeting scheduling, one of the agents is the host and the other agent is the invitee. The two-agent case is substantially simpler than the case where more than two agents are involved. The main reason for this is that in the latter case the host has to evaluate the proposals of more than one agent. Because the host has to deal with more than one invitee, the synchronisation of the negotiation rounds is also more complex.

The negotiation in the two-agent case proceeds as follows. Each round the host requests the invitee to accept the proposed meeting time or to counterpropose with the earliest possible meeting time consistent with the constraints of the invitee. After the counterproposal has been made, the host evaluates it. In case an agreement has been reached, the negotiation is ended and a confirmation of the agreement is communicated to the invitee. In case an agreement has not yet been reached, the host starts a new round in the negotiation. In this new round, the host proposes to meet at a time t that satisfies three conditions. First, t should be the same or a later time than the meeting time t' proposed by the invitee in the previous round ($t' \leq t$). Secondly, time t should be the earliest feasible meeting time for the host that meets the first condition $t' \leq t$.

And thirdly, the proposed time t must be consistent with the host's constraints and preferences.

Recall that the strategy of the participants of the negotiation is to request the earliest possible meeting time. To formalise this request, we introduce the predicate $epmeet(MeetId, PosTime, L, Att, T)$. This predicate states that time $PosTime$ is the earliest time after time T such that the meeting $MeetId$ of Length L and with the set of attendants Att can be scheduled. That is, if $epmeet(MeetId, PosTime, L, Att, T)$ holds, the meeting of length L with attendants Att cannot be scheduled between T and $Postime$, but the meeting can be scheduled at time $PosTime$. Formally, $epmeet(MeetId, PosTime, L, Att, T)$ is defined as follows:

$$\begin{aligned}
&epmeet(MeetId, PosTime, L, Att, T) \leftrightarrow \\
&\quad meet(MeetId, PosTime, L, Att) \wedge PosTime \geq T \wedge \\
&\quad\quad (\forall\, T' \cdot T \leq T' < PosTime \rightarrow \neg meet(MeetId, T', L, Att))
\end{aligned}$$

The host initiates and invites the other agent to meet at a time T acceptable to the host of the meeting by means of the goal

$\mathtt{invite}(Invitee, MeetId, T, L, Att)$

The plan (rule) to achieve this goal is given next. The first two steps in the plan (the body of the rule below) which consist of the request and offer actions constitute the *negotiation phase* of the plan. In this phase, the host first requests the invitee to meet at the earliest possible time after time T that is consistent with its constraints. The invitee in response calculates a counterproposal which it communicates to the host. In reply, the host offers a meeting time T' which is the earliest possible meeting time at which the host is able to meet and which is later or equal to the time proposed by the invitee. The latter part of the plan is called the *evaluation phase* of the plan. In this phase the host evaluates the proposal of the invitee. If the time T' that is proposed by the host is identical to the original time T requested by the host, an agreement has been reached. In that case, the host confirms that an agreement has been reached. In the other case, the host starts a new round of negotiation with a request to meet at time T'.

```
invite($Invitee, MeetId, T, L, Att$) ← true |
    req($Invitee, \exists\, T1 \cdot epmeet(MeetId, T1, L, Att, T)$);
    offer($Invitee, meet(MeetId, T', L, Att)$);
    IF $T = T'$
        THEN tell($Invitee, confirm(MeetId, T)$)
        ELSE invite($Invitee, MeetId, T', L, Att$)
```

The $\mathtt{reply}(Host)$ goal is used by the invitee to negotiate with the host. The invitee first calculates an offer that it counterproposes in reply to the request of the host to meet at the earliest time possible. After a counterproposal has been established, the invitee replies to the host with a request to accept its counterproposal. In case the host confirms that an agreement has been reached, the invitee accepts the confirmation (this is implemented by the ask branch of the nondeterministic plan for `reply`).

```
reply($Host$) $\leftarrow$ true |
    begin
        offer($Host, meet(MeetId, U, L, Att)$);
        req($Host, \exists\, U1 \cdot epmeet(MeetId, U1, L, Att, U)$);
        reply($Host$)
    end+
    ask($Host, confirm(MeetId, U)$)
```

To understand the relation between the `invite` and the `reply` goals, it is most important to understand the relation between the variables T, T', and U in these goals. For this purpose, one should realise which communication statements in the goals are related. In both goals, the first two communication statements and the latter two communication statements are linked. The variable U in the offer action in the `reply` goal is computed by computing the earliest possible meeting time after T as requested by the `invite` goal. We thus obtain $U \geq T$. Consecutively, the variable T' is computed likewise as the earliest meeting time possible after U. We thus obtain $T' \geq U$. Combining these two relations we get $T \leq U \leq T'$, and we see that progressively later times are proposed by the host and the invitee, respectively.

Note that the host as well as the invitee only make offers for meeting times which are consistent with their constraints and preferences. For example, if an agent already has scheduled to attend another meeting at time T, the agent will not offer this time in reply to a request since it violates constraint 2 of section 6.1. The consistency condition present in the abductive semantics of offer and req enforces these integrity constraints automatically.

Also note that the synchronisation of the negotiation rounds is achieved by using *synchronous* communication primitives. Because both req as well as offer are *blocking* communication primitives, the host and the invitee can only proceed to a next negotiation round together. Thus, each of the two communicative actions in a negotiation round has to be finished by both the host and the invitee before a next round can start.

6.4 Using tell and ask for Meeting Scheduling

In the previous section, a protocol for two-agent meeting scheduling was designed by using the primitives req and offer. The question we address in this section, is whether we could just as well have used the other set of communication primitives for the exchange of information to program two-agent meeting scheduling. The question thus is whether or not the intuitive reading of the primitives actually helps the programmer in making a choice concerning the use of either set of primitives. We also want to evaluate the use of the deductive versus the abductive semantics.

We will argue that the use of the req and offer in the meeting scheduling example is more natural. Of course, it is possible to write a program for meeting scheduling which only uses the tell and ask primitives, but we will try to convince the reader that these solutions are less intuitive and less elegant. Since we cannot give a formal proof of our claim, we argue by comparison and discuss a few attempts to provide a programming solution for two-agent meeting scheduling by just using tell and ask. These attempts will also illustrate some of the differences between the two sets of communication primitives. After a few attempts that do not work, we present an implementation of the multi-stage negotiation protocol that only uses the tell and ask primitives and discuss some of the disadvantages of this implementation.

The program for meeting scheduling of the previous section is quite a simple and concise solution, and therefore it seems natural to look for a similar program which uses tell and ask instead of req and offer. To find a program for meeting scheduling which just uses tell and ask, we will therefore use the strategy of finding particalur tell and ask actions which can serve to replace the req and offer in the plans provided in the previous section. The idea, thus, is to look for tell and ask actions which - if the req and offer actions used in the solution of the previous section are replaced by the tell and ask actions respectively - results in an implementation of the protocol that is correct.

The strategy of looking for tell actions to replace req actions, and ask actions to replace offer actions cannot be reversed in the sense that req actions are replaced by ask actions and replacing offer actions with tell actions. The reason is that the host is supposed to make the first proposal in the negotiation protocol presented and the invitee does not have the relevant information concerning meeting times which are acceptable to the host to make a first proposal.

First Attempt:
The first and most naive attempt would be to try and implement the requests of the agents by

$$\mathsf{tell}(a, meet(MeetId, T, L, Att))$$

and the offers of the agents with

$$\mathsf{ask}(a, epmeet(MeetId, T1, L, Att, T))$$

where the agent name a is substituted appropriately by either the host name *Host* or the invitee name *Invitee*. The main problem with this solution is that the proposition $meet(MeetId, T, L, Att)$ which the agents communicate may be inconsistent with the belief base of the receiving agent! There is no consistency check included in the semantics of the tell and ask primitives. Moreover, the meeting time proposed by the host will be accepted immediately since $T1 = T$ always provides an answer to the question asked. In a sense, we can conclude that no useful computation has been done by this solution other than the receiving of the proposed time by means of the communicative actions.

Second Attempt:
The second attempt to implement the negotiation between host and invitee is to introduce a new predicate $proposal(MeetId, T, L, Att)$ which is not used for other purposes and implement the protocol by using this predicate. The advantage of using this new predicate, of course, is that now we no longer have to worry about inconsistencies. The basic idea of the protocol is that the receiving agent either accepts or proposes a counterproposal, so we introduce a new predicate

$$count\_prop(MeetId, T', L, Att)$$

which the receiving agent will use to derive a counterproposal given a proposal of the other agent. The idea thus is to replace req actions with a tell action of the form $\mathsf{tell}(a, proposal(MeetId, T, L, Att))$ and offer actions with actions of the form $\mathsf{ask}(a, count\_prop(MeetId, T', L, Att))$.

Of course, for this idea to work, we need to define a logical relation between the predicates *proposal* and *count_prop*. Because we want the receiving agent to derive a counterproposal from a received proposal, we need to state the conditions which allow the derivation of a counterproposal from a given proposal. Intuitively, we thus need to state these conditions in the antecedent of a clause with consequent $count\_prop(MeetId, T', L, Att)$. A counterproposal must satisfy at least two conditions. First, the time T' that is counterproposed must be later than time T of the given proposal (in order to obey the strategy of the participants outlined above). Secondly, the time T' that is counterproposed must be the earliest time possible after the proposed time T. The following clause therefore seems both intuitive and plausible:

$$\begin{array}{l}(proposal(MeetId, T, L, Att) \ \wedge T' \geq T \wedge \\ (\forall\, T1 \cdot T \leq T1 < T' \rightarrow \neg meet(MeetId, T1, L, Att))) \\ \qquad \rightarrow count\_prop(MeetId, T', L, Att)\end{array}$$

The main problem with this clause, however, is, as in the first attempt, that it is possible to prove that a proposal $proposal(MeetId, T, L, Att)$ is accepted straight away. The point is that by choosing $T' = T$ (such that the time that is counterproposed equals proposed time) it is always possible to derive the counterproposal $count\_prop(MeetId, T', L, Att) \wedge T' = T$ (since the antecedent of the implication in that case trivially follows). The consequence of this solution thus is that again no useful computation other than the receiving of the proposed time has been achieved, and some other part of the program needs to compute a time for a counterproposal.

A Solution:
By analysing what went wrong in the previous attempt we can come up with a working solution. The main problem with the second attempt is that the incorporation of the condition that there exists no earlier possible meeting time in the antecedent did not work. In fact, the attempt also suffered from the problem that all constraints in the belief base of the agent might be violated. Furthermore, we need to *compute* a time T' which is the earliest possible meeting

time instead of assuming such a time is given (as in the previous attempt in the antecedent).

The solution that we present now again will make use of the predicates $\mathit{proposal}(\mathit{MeetId}, T, L, \mathit{Att})$ and $\mathit{count\_prop}(\mathit{MeetId}, T', L, \mathit{Att})$. The idea is the same as in the previous attempt and requests are to be replaced by

$$\mathsf{tell}(a, \mathit{proposal}(\mathit{MeetId}, T, L, \mathit{Att}))$$

where a is substituted appropriately by either the host name *Host* or the invitee name *Invitee*. The offers, however, cannot simply be replaced by the action $\mathsf{ask}(a, \mathit{count\_prop}(\mathit{MeetId}, T', L, \mathit{Att}))$ as was shown above. We need to find another, more complex formula to replace the *count_prop* proposition. We also need to redefine the logical relation between the *proposal* and *count_prop* predicates. We propose the following definition:

$$\mathit{proposal}(\mathit{MeetId}, T, L, \mathit{Att}) \wedge T' \geq T \rightarrow \mathit{count\_prop}(\mathit{MeetId}, T', L, \mathit{Att})$$

Thus a counterproposal can be derived from a proposal if it proposes a meeting time T' later than or equal to that of the given time T of the proposal. The implication codes part of the strategy that agents use to make counterproposals. That is, the strategy to counterpropose a *later* time than the one proposed.

The conditions formulated in the antecedent for deriving a counterproposal, however, are obviously too weak. A counterproposal may still violate the integrity constraints associated with meetings. To remedy this, the receiving agents need to code these constraints in their questions which replace the offer actions in the program of the previous section. By incorporating the integrity constraints in a slightly changed form in the questions, we can deduce that the time T' that is counterproposed meets all the constraints. We simply check whether the proposed time T' satisfies the constraints one by one in the question. The ask action that we are looking for then looks like this:

$$\begin{array}{l}
\mathsf{ask}(\mathit{count\_prop}(\mathit{MeetId}, T', L, \mathit{Att}) \wedge \\
\quad (\forall\, T1 \cdot T \leq T1 < T' \rightarrow \neg \mathit{meet}(\mathit{MeetId}, T1, L, \mathit{Att})) \wedge \\
\quad (\forall\, \mathit{MeetId}1, T1, L1, \mathit{Att}1 \cdot (\mathit{meet}(\mathit{MeetId}1, T1, L1, \mathit{Att}1) \wedge \\
\qquad\qquad T' \leq T1 < T' + L) \rightarrow \mathit{Att}1 \cap \mathit{Att} = \varnothing) \wedge \\
\quad 0:00 < L < 8:00 \wedge \\
\quad T' \geq 8:00 \wedge T' + L \leq 18:00 \\
)
\end{array}$$

As can easily be checked, the three latter conjuncts correspond to the integrity constraints (2), (3), and (4) introduced earlier. We do not need to list the first constraint, since we assumed that the meeting identifier *MeetId* is a new and unique one. Also note that the constraint $0:00 < L < 8:00$ can be dropped (the negotiation should not have been started if the length of the meeting that is being negotiated violates this constraint).

Although we believe this solution works, it has a number of disadvantages. The main disadvantage is that the integrity constraints have to be explicitly mentioned everywhere in the plans of agents and a more general approach to deal with integrity constraints is sacrificed for a reduction of the number of communication primitives. The simplicity of the solution given in terms of req and offer in the previous section is also lost. We believe this shows that the new communication primitives provide additional expressivity that is useful and moreover is not provided by the tell and ask primitives.

6.5 Binary Semaphores

The two-agent meeting scheduling problem was relatively simple because the host only had to negotiate with one other agent. The restriction to two agents especially simplified the problem of synchronising negotiation rounds. We would like to generalise the two-agent solution to a solution for arbitrary numbers of agents. The most simple idea to extend the two-agent solution to a multi-agent solution of more than two agents seems to be to let the host negotiate with all the other agents simultaneously (in parallel, that is). Each negotiation round the host then has to manage n subgoals for negotiating with all the invitees. In that case, the basic problems that have to be solved are the problem of shared resources and the problem of synchronising multiple subgoals (of the host). The solution that we propose is to use *semaphores*. In this section, we show how we can implement semaphores in 3APL.

A *semaphore* is a facility well-known from concurrent programming (Andrews 1991). One of the problems which arises in concurrent programming is how to manage different concurrent processes which access some shared object. The part of a concurrent program which accesses the shared object is called a *critical section*. Semaphores are used to guarantee mutual exclusion of critical sections. Interferences between critical sections which share some object is prevented in this way. A semaphore has two associated operators, the $\mathbf{P}(s)$ and $\mathbf{V}(s)$ operators. The $\mathbf{P}(s)$ operator decreases the semaphore s until some lower bound is reached. The $\mathbf{P}(s)$ acts like a guard and allows a process to enter its critical section only if the lower bound on the semaphore has not yet been reached. The $\mathbf{V}(s)$ semaphore increments the semaphore, thereby indicating that other processes may use the shared resources again. In particular, a binary semaphore allows that at most one process is executing its critical section at a time. As an aside, we remark here that binary semaphores can be used to implement arbitrary semaphores (cf. Andrews (1991)).

For our purposes, *binary* semaphores are sufficient. To implement semaphores in 3APL, we use the synchronous communication primitives tell and ask. The idea is to introduce a new agent sem_a for each agent a which keeps track of the semaphores that are used by agent a. Such an agent is called a *semaphore agent for* a and goes by the name sem_a. To keep track of the semaphores used by agent a, a semaphore agent sem_a adopts one goal for each binary semaphore s that is used by agent a.

Although our informal explanation of a semaphore explained the workings of a semaphore as an entity that is being raised and lowered, in our implementation of a semaphore we will not use variables that are increased and decreased. Instead, a propositional symbol p_s will be used by the semaphore agent with the informal interpretation that the critical section may be entered. An agent that wants to enter its critical section that is guarded by a semaphore s will ask its semaphore agent to confirm p_s. An agent that exits a critical section will tell its semaphore agent p_s. The informal explanation of raising and lowering a semaphore, however, can still be used to understand the way a semaphore works.

The semaphore agent sem_a continually executes a goal $semaphore_s$ for each semaphore s. The following plan rule provides a plan to implement semaphore goals and should be included in the rule base of the semaphore agent sem_a:

```
$semaphore_s \leftarrow \mathsf{true}$ |
    $\mathsf{tell}(a, p_s)$;
    $\mathsf{ask}(a, p_s)$;
    $semaphore_s$
```

The $\mathbf{P}(s)$ and $\mathbf{V}(s)$ operators which are used by an agent then can be defined by:

$$\mathbf{P}(s) \equiv \mathsf{ask}(sem_a, p_s)$$

$$\mathbf{V}(s) \equiv \mathsf{tell}(sem_a, p_s)$$

It is not hard to see that s is a binary semaphore, and only one goal is allowed to enter a section guarded by the semaphore s (by using $\mathbf{P}(s)$ and $\mathbf{V}(s)$) at a time. The tell in the semaphore goal can be viewed as a stopping sign which may only be passed if no other goal is currently in a critical section. The $\mathbf{P}(s)$ command waits until the semaphore 'tells' it it may enter its critical section. When leaving its critical section, the goal should again 'tell' the semaphore goal that it is allowed to let other goals enter their critical section. This is realised by implementing the $\mathbf{V}(s)$ command as a tell action.

6.6 Multi-agent Meeting Scheduling

Meeting scheduling in the case of more than two agents is considerably harder than for two agents. The host of the meeting now has to contact all the invitees to the meeting, and has to compute a common meeting time from the proposals for meeting times that the invited agents have made.

Basically, our task is to rewrite the `invite` plan of the host such that it can deal with any number of agents. (As we will see, the `reply` plans can be reused.) For this purpose, we assume that the host has a *list of agents* that will be invited for the meeting. With each of these agents, the host has to negotiate about a feasible meeting time. The main negotiation goal thus can be split up and each of the individual negotiations with invited agents can be viewed as a subgoal of the host's goal of negotiating a meeting time with the group of invitees. Accordingly, in the host agent, we will create a number of subgoals corresponding to the number of invitees. For this purpose, we use the post operator defined in chapter 4 for posting a goal.

For administrative purposes, we introduce a number of new predicates. These predicates are used to keep track of a number of parameters in each of the negotiation rounds and to conclude that an agreement has been or has not yet been reached. First, the predicate $count(Set, N)$ returns the number N of items in a set Set. It is used to compute the number of participants of a proposed meeting.

The strategy that the agents use is the same as that used by the agents in the two-agent case. Each agent proposes the earliest time at which - as far as that agent is concerned - the meeting can be scheduled. Each round of the negotiation the host requests all the invitees to respond to his proposal for a meeting time. A predicate $invitee(MeetId, N)$ is introduced to keep track of the number of invitees that have responded so far in a round to the proposal of the host. If the number is the same as the number of invitees, this indicates that the host may start a new round in the negotiation. A predicate $disagree(MeetId, N)$ is used to count the number of agents in a round which (still) disagree with the proposal of the host. If this number is 0, an agreement has been reached, and the negotiation is finished. A predicate $bestproposal(MeetId, Time, Length)$ is used by the host to store the best proposal made so far in a negotiation round. The best proposal is identified with the proposal which schedules the meeting at the latest time.

Remark 6.6.1 Before we present the negotiation plans of the host, we make a few remarks about the update semantics of the action ins. Needless to say, we assume that φ is implied if $\mathsf{ins}(\varphi)$ is executed (successfully). However, we will also require that the set of integrity constraints is never violated by the execution of an action $\mathsf{ins}(\varphi)$. To insert a new belief into the belief base, it thus may be necessary to remove some of the current beliefs to maintain the integrity constraints, and here we diverge from the earlier definition of ins presented in chapter 3. Furthermore, we assume that the changes made to the belief base are *minimal* in a some suitable sense (cf. Gärdenfors (1988)).

Finally, we will assume that the following constraint holds for all predicates p from the set of predicates $\{invitee, disagree, bestproposal\}$ that were introduced above:

$$(p(x_1, \ldots, x_n) \land p(y_1, \ldots, y_n)) \rightarrow x_1 = y_1 \land \ldots \land x_n = y_n$$

This constraint states that at most one tuple can have property p at a time and implies that for an $\mathsf{ins}(p(t_1, \ldots, t_n))$ action to be successful it has to remove any current beliefs about p before a new belief $p(t_1, \ldots, t_n)$ can be inserted into the belief base. Moreover, for each of the predicates *invitee*, *disagree*, and *bestproposal* we assume that ins always can be successfully executed.

Now we are able to present the plans for the host to negotiate with arbitrary numbers of agents. We distinguish two types of plans. The first type of plans is used by the host to initiate and to structure the negotiation process itself. The second type of plan is used by the host to deal with the negotiation and communication between the agents.

The first plan `initiate_negotiation` consists of a number of instructions to set up the initial conditions of the negotiation and introduce a set of plans to perform the actual negotiation. The plan initialises the predicates *invitee*, *disagree*, and *bestproposal* and creates a number of subgoals. Each of these subgoals corresponds to one of the invitees and is introduced to specifically negotiate with that invitee. The subgoal `initiate_indiv_negotiation` creates these subgoals. The predicates *invitees*, *disagree* and *bestproposal* are introduced for bookkeeping purposes. Initially, non of the N invitees has agreed to a meeting time. This fact is represented by $disagree(MeetId, N)$. $invitee(MeetId, M)$ is used to represent the number of invitees that have responded to the host's proposal. Since initially none of the invitees has responded the proposition $invitee(MeetId, 0)$ is inserted into the host's beliefs. The proposition $bestproposal(MeetId, T, L)$ is inserted into the host's beliefs because initially the best proposal on the table is the initial proposal of the host itself. The subgoal `negotiation` introduces a plan that structures the negotiation.

We use the expression $Set - Item$ to denote the set obtained from removing $Item$ from Set. Below, the variable Att denotes a set of agents.

```
initiate_negotiation($MeetId, T, L, Att$) $\leftarrow$ true |
    $count(Att - Host, N)?$;
    ins($bestproposal(MeetId, T, L)$);
    ins($disagree(MeetId, N)$);
    ins($invitee(MeetId, 0)$);
    initiate_indiv_negotiation($Att - Host, MeetId, T, L, Att$);
    negotiation($MeetId, Att$)
```

The `negotiation` plan, the second plan, is used to structure the negotiation into several distinct rounds. It waits until all N invitees have responded to the request of the host to schedule the meeting at a particular time (which is dealt with by the subgoals to negotiate with each invitee individually). For this purpose, the predicate *invitee* which keeps track of the number of participants that have responded so far is used in the test $invitee(MeetId, N)?$. If all agents have responded, the `negotiation` plan checks whether an agreement has been reached. If so, $disagree(MeetId, 0)$ must hold, and the best proposal so far is retrieved and used to schedule the meeting by means of the `schedule_meeting` subgoal. In case an agreement has not yet been reached, a new round is started by reinitialising the predicates *invitee* and *disagree* for the next round.

```
negotiation($MeetId, Att$) $\leftarrow$ true |
    $count(Att - Host, N)$?;
    $invitee(MeetId, N)$?;
    IF $disagree(MeetId, 0)$
    THEN begin
            $bestproposal(MeetId, T, L)$?;
            schedule_meeting($MeetId, T, L, Att$);
        end
    ELSE begin
            ins($disagree(MeetId, N)$);
            ins($invitee(MeetId, 0)$);
            negotiation($MeetId, Att$)
        end
```

The next two plan rules are used to create the subgoals of the host to negotiate with each of the invitees individually. The post operator is used for this purpose and posts for each of the invitees an `invite` subgoal.

```
initiate_indiv_negotiation($Invitees, MeetId, T, L, Att$)
  $\leftarrow$ $Invitee \in Invitees$ |
    post(invite($Invitee, MeetId, T, L, Att$));
    initiate_indiv_negotiation($Invitees - Invitee, MeetId, T, L, Att$)

initiate_indiv_negotiation($Invitees, MeetId, T, L, Att$) $\leftarrow$
     $Invitees = \varnothing$ |
```

The `initiate_indiv_negotiation` goal creates a set of parallel `invite` goals in the goal base of the host. Each of these goals is used by the host to simultaneously deal with individual negotiations with invitees. The set up for the host thus creates a goal base which includes a goal `negotiation` to structure the negotiation and a number of goals corresponding to the number of invitees to deal with individual negotiation. All of these goals run in parallel and are synchronised by means of binary semaphores and communication through the shared belief base of the host.

The previous plans were all designed to initiate and structure the negotiation process. The last plan of the host is a plan for negotiating with each of the individual invitees. The subgoals created to negotiate with each of the invitees are `invite` goals that are similar to the two-agent case, but some changes have been made. In particular, the evaluation phase of the plan for `invite` goals in which the proposals of the invitees are evaluated has to be revised in order to consider the proposals of all of the invitees. The negotiation phase, however, has not been changed. The simple strategy that the host uses in the evaluation phase is to select the proposal that proposes the *latest* time and is also compatible with the host's own constraints as the 'winning' proposal. In case there is no such proposal, the host computes a new meeting time that is later than all proposed times and that is suitable from its own perspective. The negotiation then continues with this new tentative meeting time.

The subgoals of the host which negotiate with the invitees have to communicate with each other about the proposals they receive from the invitees. The communication between the subgoals is achieved by means of the belief base and the predicate *bestproposal*. Each of the subgoals compares its received proposal with the latest proposal for a meeting time stored in the belief base of the agent as $bestproposal(MeetId, T, L)$. It also raises a counter $invitee(MeetId, N)$, to indicate that it did receive a proposal and is waiting for the next round of the negotiation. The evaluation phase is a critical section in the plans and mutual exclusion needs to be guaranteed between the different subgoals since in these sections of the plans the predicate *bestproposal* is updated. *Binary* semaphores are used to implement the mutual exclusion of the subgoals. The $\mathbf{P}(bp)$ operator increments the semaphore, and the $\mathbf{V}(bp)$ operator decreases the semaphore.

```
invite($Invitee, MeetId, T, L, Att$) $\leftarrow$ true |
    req($Invitee, \exists T1 \cdot epmeet(MeetId, T1, L, Att, T)$);
    offer($Invitee, meet(MeetId, T', L)$);
    $\mathbf{P}(bp)$;
        $bestproposal(MeetId, TentativeTime, L)?$;
        IF $TentativeTime < T'$ THEN ins($bestproposal(MeetId, T', L)$);
        $disagree(MeetId, M)?$;
        IF $T = T'$ THEN ins($disagree(MeetId, M - 1)$);
        $invitee(MeetId, N)?$;
        ins($invitee(MeetId, N + 1)$);
    $\mathbf{V}(bp)$;
    * wait until every agent has replied and a new round begins :
    * invitee(MeetId, 0) is inserted by the negotiation plan;
    * after insertion of this fact start a new round.
    $invitee(MeetId, 0)?$;
    IF $disagree(MeetId, 0)$
        THEN tell($Invitee, confirm(MeetId, T')$)
        ELSE invite($Invitee, MeetId, T', L, Att$)
```

Finally, we repeat the plans for the invitees to reply to the host. The plans for the `reply` goals of the invitees do not have to be changed and are still applicable.

```
reply($Host$) $\leftarrow$ true |
    begin
        offer($Host, meet(MeetId, T, L, Att)$);
        req($Host, \exists T1 \cdot epmeet(MeetId, T1, L, Att, T)$);
        reply($Host$)
    end+
    ask($Host, confirm(MeetId, T)$)
```

6.7 Conclusion

We illustrated the use of the new communication primitives by an extended example concerning meeting scheduling agents. We implemented a multi-stage negotiation protocol, both for the case where only two agents are involved in the negotiation and the case where an arbitrary number of agents are involved in the negotiation. The implementation is both natural and concise. The latter illustrates the expressive power of the communication primitives, while the former shows that the agent programming language 3APL is a natural means for programming personal assistants. Furthermore, we implemented binary semaphores by using the tell and ask primitives. These were used in the implementation of the multi-stage negotiation protocol for an arbitrary number of agents.

Part II:

Comparing Agent Languages

Agent-based computing in Artificial Intelligence has given rise to a number of diverse and competing proposals for agent programming languages. Agents, in the sense we are using it, are computational entities with a mental state consisting of components like the beliefs, the goals, and the intentions of that agent. A number of languages are based on this notion or based on similar programming concepts. In particular, the agent languages AGENT0 (Shoham 1993), AgentSpeak(L) (Rao 1996*a*) and ConGolog (Giacomo et al. 2000) are based on concepts that are similar to those of 3APL.

At first sight, it is not so clear, however, how these various programming languages are related to one another. There are several reasons why it has been difficult to evaluate and compare the relative benefits and disadvantages of different systems for programming agents. One of the main reasons, in our opinion, is the lack of a general semantic framework which provides a suitable basis for language comparison. Moreover, it is not so easy to compare agent programming languages because of the fact that they do not all have a formal semantics.

The operational semantics that we have been using, however, does provide a basis for such comparison. Transition systems can be used both for the specification of a semantics for programming languages which do not yet have a formal semantics, as well as for a formal comparison of languages which are formally defined. It is our aim in this second part of the thesis to study and compare the four agent languages mentioned above. AGENT0, which does not come with a formal semantics, is analysed and provided with a formal operational semantics using a transition system. As for the other languages that do have a formal semantics, we develop a general method for comparing these programming languages. The method is based on the concept of a *bisimulation*. As it will turn out, the four languages AGENT0, AgentSpeak(L), ConGolog and 3APL form a family of closely related programming languages for agents.

CHAPTER 7

An Operational Semantics for AGENT0

AGENT0 is an experimental programming language for programming *intelligent agents* that has been designed by Shoham (Shoham 1993). An intelligent agent in AGENT0 is an entity with a mental state, consisting of the *beliefs* and *commitments* of the agent, that is capable of interacting with its environment and deciding what to do. So-called *commitment rules* provide the basic means for decision-making and the introduction of new commitments.

In this chapter, we design an operational semantics for AGENT0 which is formal and facilitates a precise comparison with other agent languages. This semantics provides a detailed analysis of AGENT0 and - we believe - enhances our insight into agents and agent programming in general. Although our main focus is on providing a formal semantics which provides a basis for comparison with other languages and with clarifying the notion of agent programming in general, we also discuss some of the more methodological issues raised in Shoham's papers (Shoham 1991, Shoham 1993, Shoham 1994). In particular, we discuss why it is important to specify a formal semantics for an agent programming language and comment on the relation between the logic defined in (Shoham 1993, Shoham 1991) and the language AGENT0.

The chapter is organised as follows. First, we motivate the need for a formal semantics for the *programming language* AGENT0 and argue that such a semantics cannot be derived from the *modal logic* for reasoning about beliefs, choices, etc. as introduced in (Shoham 1993, Shoham 1991). In section 7.2, we give an informal overview of AGENT0. In section 7.3, a subset of AGENT0 is identified for programming single agents. This subset is called the *single agent core of AGENT0.* It is for this subset that we provide a formal semantics. In section 7.4, we then present a formal, operational semantics for AGENT0. This semantics specifies the meaning of the basic constructs in the language. Finally, in section 7.5, we discuss the AGENT0 interpreter and in particular the decision-making of agents in AGENT0. Our account reveals an interesting difference in the style of decision-making between AGENT0 agents and agents programmed in AgentSpeak(L) or 3APL.

7.1 Defining The Agent-Oriented Paradigm

The programming language AGENT0 supports the construction of agent programs. In (Shoham 1993), agents are defined as entities "whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments." AGENT0 supports the construction of such agents by offering programming constructs which are viewed as the formal counterparts of these mental concepts. The programming constructs stand in rough correspondence to their common sense counterparts. That is, the goal is to obtain a close enough resemblance to be suggestive and useful when programming agents, or, as Shoham puts it, to "make engineering sense out of these abstract concepts" (Shoham 1991).

The emphasis on mental states in agent-oriented programming makes it imperative to state precisely and explicitly what such a state is. For this reason, part of the agent programming paradigm in (Shoham 1991, Shoham 1993, Shoham 1994) is to provide a formal semantics of the mental state of agents. The means by which the mental terminology is made precise in these papers is provided by a formal *modal logic*. This approach is also used to define the mental components in the programming language PLACA (Thomas 1993), a successor of AGENT0.

The modal logic used to define the modal components of AGENT0 agents has been described in most detail in (Shoham 1991). The basis for the modal logic is an "explicit-time point-based logic". This temporal logic is extended with a KD45 operator (Chellas 1980) for modelling *belief* and a KD4 operator for modelling *commitment*. In terms of these operators, a *choice* of an agent is defined as a commitment to itself, and the capability to (see to it that) X (where X is a proposition) is defined as a conditional: if the agent chooses to X, then X should be the case.

In (Shoham 1993), an informal account is provided of the agent *programming language*. The features of agents written in the programming language include constructs for referring to time points, beliefs, capabilities, actions, commitment rules for committing to actions, and communication. These constructs only roughly correspond to the operators of the logic. There are more differences than similarities. To name only a few, the programming language includes an explicit representation of actions whereas the logic does not, commitments are defined in terms of these actions in the programming language (chosen actions) whereas commitment is a primitive notion in the logic, and the dynamics of the execution of agents is not modelled in the logic but is implemented by commitment rules in the programming language.

For these and other, similar reasons, there is no clear link between the modal logic and AGENT0. The formal semantics of the logic, therefore, contrasts with the lack of a formal semantics for the agent programming language itself. The situation here is analogous to that of definitions of agent communication languages where (i) a logical semantics is defined which is supposed to define the semantics of the communication language but (ii) it is not clear how this semantics is related to an operational semantics for computational agent systems (cf. also section 5.5).

There are a number of reasons why it is important to specify a formal semantics which precisely defines the meaning of the *programming* constructs in AGENT0, and other agent languages. First of all, the construction of a formal semantics requires a detailed analysis and may reveal inconsistencies or gaps in informal accounts of software systems. As our analysis will show, there are several such gaps in the informal account of AGENT0. Such an incomplete specification may give rise to ad hoc implementations that differ in important aspects. Secondly, a formal semantics explains in a precise and explicit manner the meaning of the programming language constructs and thereby enhances our understanding of agent programming and agents in general. It also provides a formal definition of the rule-based decision-making and other features of agents. A formal semantics thus facilitates the programming of agent systems. Third, a formal semantics allows for a detailed comparison with other agent programming languages. In particular, the proposal of a formal semantics for AGENT0 highlights an interesting difference in the decision-making of AGENT0 agents and that of agents written in AgentSpeak(L) (Rao 1996*a*) and 3APL. Finally, a formal semantics is a first step towards a *programming logic* for reasoning about agent programs written in AGENT0.

7.2 Overview of AGENT0

Agents are controlled by *agent programs*. In AGENT0, these programs are executed by an interpreter which continuously executes a *control loop* consisting of two phases: A phase in which the mental state is updated and a second phase in which actions (current commitments) are executed. In the first phase, messages that are received from other agents are processed and the commitment rules associated with an agent are fired. Messages can change both the beliefs as well as the commitments of an agent. By firing commitment rules, new commitments may be introduced without changing the commitments that already have been made.

We will now give a somewhat more detailed overview of the constructs that are included in the language AGENT0 to program agents. First of all, the *beliefs* of an agent are simple, time-stamped, atomic formulas, also called *facts*, from a first order language with *explicit time*, written like `(1march/10:00 (employee(John)))`. The programming language has a number of different types of *actions*. The first, most basic type is a so-called *private action* and is written like `(DO 18april/9:00 issue_boarding_pass)`. Note that these actions are also time stamped. The time indicates the time associated with the execution of the action (if it is executed). Three *communication* actions are provided. An action to inform an agent of a fact, written like `(INFORM 1march/2:00 smith (18april/10:00 (flight sf ny #293)))` which informs Smith on 1 march at 2:00 that on 18 April 10:00 a flight with number 293 from San Francisco to New York is scheduled. Two more communication primitives, the `REQUEST` and the `UNREQUEST` primitives, are supported. These primitives allow an agent to send requests to perform an action to another agent and to request an agent to drop a commitment to a particular action. Finally, conditional actions of the form `(IF mntlcond action)` where `mntlcond` is a condition on the mental state and refrain actions of the form `(REFRAIN action)` can recursively be constructed from more simple actions.

Commitments of an agent consist of actions that an agent has chosen to perform at a particular time. The decision to perform an action is regulated by *commitment rules* of the form `(COMMIT msgcond mntlcond (agent action))` where `msgcond` is a condition on the received messages and `mntlcond` is a condition on the current mental state of an agent. The rule can be fired if both conditions are satisfied. Conditions on the mental state are boolean combinations of simple conditions on the beliefs of the form `(B fact)` and simple conditions on the commitments of the form `((CMT agent) action)`. The agent that is associated with the action `action` is the agent to which the commitment is made. Conditions on the messages are boolean combinations of simple message conditions of the form `(agent INFORM fact)` or `(agent REQUEST action)`.[1] Finally, these conditions may include a number of different types of variables. As an example, the commitment rule

```
(COMMIT (?a REQUEST ?action) (B (now (myfriend ?a))) (?a ?action))
```

can be fired if a request to perform `?action` from agent `?a` has been received and the agent believes that `?a` is a friend. The constant `now` is a special constant referring to the current time. Upon firing, the agent commits to action `?action` and records that the commitment is made to agent `?a`. Thus, commitment rules provide a mechanism for making decisions in AGENT0. In the interpreter for AGENT0, this decision-making occurs in the first phase.

To summarise, the agent programming language AGENT0 includes features for representing the domain of interest (beliefs), to perform actions (simple, conditional and refrain actions), for interacting with agents (communication primitives), and for making commitments (by means of commitment rules). Each agent has it own associated set of capabilities, and actions and beliefs are explicitly associated with a particular time.

[1]It is not so clear why message conditions of the form `(agent UNREQUEST action)` are not allowed. Message conditions of this form would allow an agent to commit to an alternative plan of action, in case it is requested to drop a commitment but it still wants to achieve the goal associated with the commitment.

7.3 The Single Agent Core of AGENT0

The number of features present in the language AGENT0 is somewhat overwhelming. It is hard to understand the interaction between so many features and difficult to program with no clear understanding of the meaning of so many constructs. For simplicity, we therefore define a semantics for a subset of AGENT0 constructs. This semantics is a first approximation to a semantics which includes all features. The subset we consider in this chapter includes all features except for multi-agent communication and explicit reference to time. We call this subset the *single agent core of AGENT0*.

Remark 7.3.1 *(an interaction problem)*
One of the possible conflicts that might arise because of the interaction of several features in AGENT0 is a problem due to the combination of time stamped actions and time stamped beliefs. Although Shoham is not too clear about the semantics of actions, the idea of assigning an update semantics to actions (as we do below) runs immediately into problems due to the time stamping of actions and beliefs. Because of this time stamping, it is not possible to specify the semantics of actions individually and specify the joint execution of a number of actions by an interleaving semantics. Instead, we need to take into account the effect of arbitrary sets of actions executed at a particular moment.

As a simple example, consider an AGENT0 agent that has scheduled the actions `move_north` and `move_south` to be executed both at time `t+1` and represents its position at time `t` by the predicate `(t (loc(x,y))`. We are interested in the effect of executing both actions at time `t+1` on the beliefs of the AGENT0 agent. In case it would make sense to specify the semantics of each action separately, it should be possible to compute the effect of executing both at the same time from the specifications for each of the two actions. Given that the position is represented by `(t loc(x,y))`, a simple idea would be to define both `move_north` and `move_south` as updates that replace `(t (loc(x,y))` in the belief base with, respectively, `(t+1 (loc(x,y+1))` or `(t+1 (loc(x,y-1))`.

However, given these semantic definitions of the actions, it is not clear anymore how to define the semantics of simultaneously executing both actions. Informally, executing both actions simultaneously should not result in a change in the position of the agent at all. We cannot therefore choose to insert only one of the two updates; that is, choose to insert either `(t+1 (loc(x,y+1))` or `(t+1 (loc(x,y-1))` (which an interleaving semantics would do). Moreover, it is also quite obvious that we cannot insert both locations because of the resulting inconsistency of the beliefs of the agent. To solve these problems, it seems that we need to define the effect of simultaneously executing both actions from scratch and it is not possible to compute the effect from the semantics of the individual actions. The combination of time and actions thus may cause an interaction problem which is revealed by an attempt to specify a formal semantics for this combination of features.

The single agent core as we have defined it includes beliefs, capabilities, three types of action, commitments, and commitment rules. The specification of the formal semantics for this core is based on the informal explanation of AGENT0 in (Shoham 1993). Although we have tried to stay as close as possible to the intended meaning of AGENT0 constructs, our semantics is a reconstruction from the informal text in (Shoham 1993). Our strategy for defining the semantics of AGENT0 is to separate the specification of the meaning of the basic constructs - like the beliefs and rules - and the specification of an interpreter. This strategy has also been used to specify the agent language 3APL (cf. Hindriks et al. (1999*a*), and Hindriks et al. (1999*b*)). For a formal specification of an interpreter for AGENT0 we refer the reader to (Hindriks et al. 1999*b*, Hindriks et al. 1999*c*).

In the following sections, the single agent core is introduced and the syntax is formally defined. The syntax of the language is recast in a somewhat different notation. This notation serves our purposes better and is more suitable as a means for comparison of AGENT0 with other languages than the syntax of AGENT0 as presented in (Torrance 1991, Shoham 1993).

7.3.1 Beliefs

An AGENT0 agent essentially is a mental entity that operates on and manipulates a database of beliefs. The language for beliefs used in AGENT0 is a simple fragment of first-order logic, namely the set of literals (atomic statements and their negations). Beliefs are built from terms, predicates and negation. Here, following (Shoham 1993), we define a term as either a constant or a variable. These restrictions on the beliefs of an agent seem unnecessarily restrictive. Since no functions are allowed, nor Prolog-like programs, the computational expressiveness at this level is restricted to a bare minimum (and basically consists of pattern-matching). However, for our purposes this is of no real interest.

Definition 7.3.2 *(terms, atoms, literals)*
Let $\langle \mathsf{Pred}, \mathsf{Cons} \rangle$ be a signature, where Pred is a set of *predicate symbols*, and Cons is a set of *constant symbols*, and let Var be a countably infinite set of *variables*. Then the set of *terms* Term, the set of *atoms* At, and the set of *literals* Lit are defined by:

- $\mathsf{Var} \subseteq \mathsf{Term}$, $\mathsf{Cons} \subseteq \mathsf{Term}$,
- if $p \in \mathsf{Pred}$ of arity n, and $t_1, \ldots, t_n \in \mathsf{Term}$, then $p(t_1, \ldots, t_n) \in \mathsf{At}$,
- if $p \in \mathsf{Pred}$ of arity n, and $t_1, \ldots, t_n \in \mathsf{Term}$, then $p(t_1, \ldots, t_n) \in \mathsf{Lit}$ and $\neg p(t_1, \ldots, t_n) \in \mathsf{Lit}$.

7.3.2 Actions

When communication and time is left out of AGENT0, three types of actions remain: (i) *simple actions* of the form `(DO <privateaction>)`, constructed from a given set of so-called *private actions*, (ii) *conditional actions* of the form `(IF <mntlcond> <action>)`, where `<mntlcond>` expresses a condition on the mental state of the agent, and (iii) *refrain actions* of the form `(REFRAIN <action>)`.

The refrain action (`REFRAIN <action>`) is a type of action that precludes commitment to actions of the form `<action>`.[2]

First, we formally define the syntax of the most basic actions, called *private actions*, and for reasons that will become clear below we currently postpone the formal introduction of the other types of actions.

Definition 7.3.3 *(private actions)*
Let Asym be a set of action symbols. Then the set of *private actions* Pact is defined by:

$$\mathsf{Pact} = \{\mathsf{a}(t_1, \ldots, t_n) \mid \mathsf{a} \in \mathsf{Asym} \text{ of arity } n, \text{ and } t_1, \ldots, t_n \in \mathsf{Term}\}$$

The definition of actions provides an instructive example of the mismatch between the modal logic for defining the mental state and the programming language. Although in (Shoham 1993) it is stated that "in the programming language" actions are "introduce[d] [...] as syntactic sugar", actually, in the programming language an explicit construct `DO` is introduced and the so-called private actions may range from 'retrieval primitives', 'mathematical procedures' to robotic, physical actions. As an example, in the manual (Torrance 1991) of AGENT0, a number of basic or primitive actions programmed in Lisp are provided. These primitive actions are *not* propositions, but are explicit actions useful for programming agents in AGENT0. Moreover, the programming language allows complex conditional and refrain actions which have no counterpart in the modal logic. And finally, a number of communicative actions are supported by the programming language which are not present in the logic.

In the modal logic as defined in (Shoham 1991, Shoham 1993), the most primitive actions, called *private actions*, are represented by propositions. In the logic no explicit and distinct representation for actions is present nor are there any modalities for actions incorporated into the logic as in, for example, dynamic logic (Harel 1979). In (Jones 1993), in particular this feature is criticised. The specific axioms of the logic which are proposed in (Shoham 1991) turn out to have the highly counter-intuitive consequence that if an agent cannot do something, then it believes that it can do it. To secure consistency within the logic, as a consequence, an agent cannot believe that it is incapable of anything. In (Jones 1993), the main conclusion is that the core of this problem is due (in part at least) to the absence of an explicit representation of action.

For these reasons, there is a mismatch between the programming language and the modal logic in (Shoham 1993) in the representation of actions. In particular, whereas in the logic commitments are represented as obligations to a fact holding, in the programming language an agent commits to actions to change its mental state. As a consequence, the requirement that commitments to actions should be "internally consistent" makes only sense in the logic but not in the programming language. Moreover, the principle of "good faith" which requires that a commitment to see to it that a proposition holds implies that the agent believes that the proposition will hold cannot straightforwardly be

[2] According to Shoham, the refrain action "is really a non-action" ((Shoham 1993), p. 72).

translated to a statement about the programming language. Also, the persistence conditions with respect to beliefs and commitments are only discussed informally in (Shoham 1993); no formal semantic account for either the logic or the programming language is provided.

7.3.3 Variables

In (Shoham 1993) a number of different types of variables are introduced. The types of variables in AGENT0 include variables ranging over agents, beliefs, and action statements as well as first order variables. For our purposes, the first type - agent variables - are not very interesting since we focus on the single agent core of AGENT0. Therefore, we do not include these variables. The second type of variables in the list, variables ranging over beliefs, provides a kind of higher-order feature concerning information, whereas the third type provides a higher-order feature concerning actions. They are, however, not included in the BNF syntax definition in (Shoham 1993). In the absence of any complex beliefs or complex actions constructed by means of regular programming operators, we have doubts concerning the use of both types of variables. In this context, it seems to allow only for very simple pattern matching. For example, a rule could be programmed which expresses that if the agent is committed to *some* action, it should inform another agent that the agent is currently busy. In (Shoham 1993), some examples are provided of variables ranging over beliefs, but no interesting examples are offered for variables ranging over actions. The intended semantics of these variables is also not entirely clear. As far as variables are concerned, we therefore discuss only first order variables.

In (Shoham 1993), two types of first order variables are introduced, 'existentially quantified' and 'universally quantified' variables. Whereas the 'existentially quantified' variable is used similarly as variables are used in logic programming, the semantics of the 'universally quantified' variable is less clear. The use of a universally quantified variable, denoted by the prefix "?!", is illustrated in (Shoham 1993) by the following example:

`(IF (B (t (emp ?!x acme))) (INFORM a (t (emp ?!x acme)))).`

As explained in (Shoham 1993), this conditional action results in informing all employees which are believed to have acme of that fact. The scope of the 'universally quantified' variable seems to be the entire conditional.

The combination of both types of variables, however, leads to a problem concerning the order of the quantifiers. For example, what does a statement `(IF (B (friend ?!x ?y)) (INFORM a (friend ?!x ?y)))` mean? Does it mean that agent `a` should be informed in case everybody has a friend or in case there is somebody who is everybody's friend?

Because of this problem, in this chapter we only allow one type, the 'existentially quantified' variable ranging over the domain of discourse of the agent, and do not consider a 'universally quantified' variable. In PLACA (Thomas 1993), 'universally quantified' variables have been left out of the language, and in AGENT0 they were not included in the actual implementation.

The types of variables allowed in the programming language provide another example of the mismatch between the logic and the programming language. The different types of variables in the programming language do not have counterparts in the logic. From the informal text, moreover, it is not easy to reconstruct the intended semantics of the different types of variables in the programming language.

7.3.4 Actions and Mental State Conditions

An AGENT0 agent is allowed to inspect both its beliefs and its commitments for decision-making. During the execution of (conditional) actions an agent is also allowed to inspect its mental state. The beliefs of an agent are simple facts (literals) from a first order language. The commitments of an agent are the actions it has selected for execution. Together, the beliefs and commitments of an agent make up its mental state.

Since conditions on mental states may refer to actions and (conditional) actions may refer to mental state conditions, actions and mental state conditions are defined by simultaneous induction. Perhaps in our definition of mental state conditions we deviate the most from the syntax of AGENT0 as introduced in (Shoham 1993). Our reason for doing so is that we want to be as precise as possible and at the same time aim at an operational semantics which can be readily implemented. Although the logic-like notation in (Shoham 1993) is very suggestive, the meaning of the notation is not so clear. In particular, to understand the parameter mechanism of the programming language it is important to have a formal semantics. Our notation is more suited for this purpose, though it is less suggestive as the notation introduced in (Shoham 1993). We discuss these issues more extensively below when the operational semantics is defined.

Actions are defined starting with private actions and mental state conditions are defined as four-tuples consisting of: (i) a set of literals the agent should believe, (ii) a set of literals the agent should not believe, (iii) a set of actions an agent should be committed to, and, finally, (iv) a set of actions the agent should not be committed to. A mental state condition is fulfilled if all of these conditions hold.

Definition 7.3.4 *(actions and mental state conditions)*
The set of actions Act, and the set of mental state conditions $\mathsf{MentCond}$ is defined by simultaneous induction:

- The set of actions Act:
  - $\mathsf{Pact} \subseteq \mathsf{Act}$,
  - if $a \in \mathsf{Act}$ and $c_1, \ldots, c_n \in \mathsf{MentCond}$, then $(c_1, \ldots, c_n : a) \in \mathsf{Act}$, also called *conditional actions*,
  - if $a \in \mathsf{Act}$, then $\delta a \in \mathsf{Act}$, also called *refrain actions*,
- The set of mental state conditions $\mathsf{MentCond}$:

– if $L^+, L^- \subseteq \mathsf{Lit}$ and $C^+, C^- \subseteq \mathsf{Act}$,
then $\langle L^+, L^-, C^+, C^- \rangle \in \mathsf{MentCond}$.

Informally, a conditional action $(c_1, \ldots, c_n : a)$ is executed by testing whether *one* of the conditions c_i holds and, if so, continuing with the execution of action a. A refrain action δa removes commitments to actions of the form a. The action a thus specifies the type of actions that should be refrained from. Finally, a mental state condition $\langle L^+, L^-, C^+, C^- \rangle$ holds relative to a given mental state if each of the literals in L^+ is believed, none of the literals in L^- is believed, each of the actions in C^+ is committed to and none of the actions in C^- is committed to.

7.3.5 Commitment Rules

The decision-making of AGENT0 agents is implemented by so-called *commitment rules.* Commitment rules introduce new commitments. They do not remove or modify the current commitments of the agent, but simply add new ones. A commitment rule consists of two parts: (i) a condition on the mental state of an agent and (ii) an action (recall that we do not discuss communication which explains the absence of message conditions in commitment rules). The action part represents the new commitment that is to be made if the rule is fired. A commitment rule thus (almost) has the same structure as a conditional action, but for clarity and to be able to keep them apart we introduce some new notation for commitment rules:

Definition 7.3.5 *(commitment rules)*
The set of *commitment rules* CommitRule is defined by:

- if $\langle L^+, L^-, C^+, C^- \rangle \in \mathsf{MentCond}$, and $a \in \mathsf{Act}$,
then $C^+, C^- \leftharpoondown L^+, L^- \mid a \in \mathsf{CommitRule}$.[3]

7.3.6 Agent Programs

Now we have introduced the basic constructs in the language AGENT0, we are able to define what an agent (program) is. Syntactically, an agent program is a set of capabilities, a set of initial beliefs, and a set of commitment rules. The set of capabilities in the program defines the expertise of the agent. Capabilities consist of a mental state condition and a private action.[4] The condition specifies under what (mental) condition the agent is capable of executing the action. The initial beliefs specify what the agent believes, at the start of execution. Finally, the commitment rules determine what types of decisions - new commitments - the agent will make. Note that initially the set of commitments is supposed to be empty. It is not clear from (Shoham 1993) why this is required.

[3]In the BNF syntax of AGENT0, multiple actions are allowed in a rule instead of a single action a. Since this feature can be simulated by rules with a single action, however, we have restricted rules to the more simple format with a single action in the body.

[4]In the BNF grammar of AGENT0 in (Shoham 1993) every type of action is allowed. In the main text (p. 77) only private actions are allowed.

Definition 7.3.6 *(agent program)*
An *agent program* is a tuple $\langle \mathsf{Cap}, \sigma_0, \Gamma \rangle$, where

- Cap is a set of *capabilities*, i.e. a set of actions of the form $(c_1, \ldots, c_n : a)$, where $c_i \in \mathsf{MentCond}$ for all i and $a \in \mathsf{Pact}$,
- $\sigma_0 \subseteq \mathsf{Lit}$ is the set of *initial beliefs*, and
- $\Gamma \subseteq \mathsf{CommitRule}$ is a set of *commitment rules*.

7.4 An Operational Semantics for the Core of AGENT0

In this section, we define a formal semantics that specifies the meaning of the single agent core of AGENT0. The semantics of mental state conditions, actions and commitment rules is defined. The semantics in this section is based on the informal account of the programming language in (Shoham 1993) and only discusses the logical approach when it offers a different perspective or there is a difference between the two. Because the informal account is not always precise or detailed enough, there are a number of gaps in the account in (Shoham 1993) which we had to fill in to specify a semantics for AGENT0.

7.4.1 Transition Systems

The semantics we provide for AGENT0 is an *operational* semantics. For this purpose, we use a *transition system* which defines the computation steps an AGENT0 agent can perform. In agent-oriented programming, the notion of a mental state is the basic concept. Agent programs can be viewed as transition functions on mental states. The transition relation defined by the transition system is a relation on mental states. It specifies how computation steps of an agent program transform mental states. Because the commitment rules and the agent's capabilities do not change during execution, we do not include them in the mental state and will not mention them explicitly anymore.

Definition 7.4.1 *(mental state)*
A *mental state* is a pair $\langle \Pi, \sigma \rangle$, where $\Pi \subseteq \mathsf{Act}$ is a set of actions, also called *commitments*, and σ is a set of *beliefs*.

We assume that there are no occurrences of free variables in an agent's belief base (cf. also chapter 3).

7.4.2 Semantics of Mental State Conditions

In (Shoham 1993), no (informal) explanation is given of the semantics for mental state conditions. The use of a logic-like notation suggests that the formal semantics of the modal logic used in (Shoham 1993, Shoham 1991) should fill in this gap. However, the logic does not provide an appropriate account. In particular, the parameter mechanism and scope of free variables in the programming language is not explained by the logic.

Our semantics of mental state conditions is directly derived from the mental state of an agent as defined in definition 7.4.1. Although we cannot be sure that this semantics fully corresponds to that of the intended semantics, it probably provides a good approximation and completes the specification of the semantics for the language. Moreover, it is a precise semantics which can be evaluated on its merits and deficiencies. The semantics of beliefs is derived from the semantics of first order logic; we use $\models$ to denote the usual consequence relation of first order logic. To specify a parameter mechanism for AGENT0 that is used to compute bindings for free variables, we use the notion of a substitution. A substitution is a finite set of pairs (also called bindings) of variable-term pairs (for a more explicit and formal definition, see chapter 3).

Definition 7.4.2 *(semantics of mental state conditions)*
Let θ be a substitution. A mental state condition $c = \langle L^+, L^-, C^+, C^- \rangle$ is *satisfied* in a mental state $M = \langle \Pi, \sigma \rangle$ relative to θ, notation: $M \models c\theta$, if:

- for each $\varphi \in L^+$, we have that $\sigma \models \varphi\theta$,
- for each $a \in C^+$, we have that $a\theta \in \Pi$,
- for all $\varphi \in L^-$ and *all* substitutions γ we have that $\sigma \not\models \varphi\gamma$, and
- for all $a \in C^-$ and *all* substitutions γ we have that $a\gamma \notin \Pi$.

Thus, a mental state condition is satisfied if (i) it is possible to instantiate the free variables in L^+ and C^+ uniformly such that the agent's beliefs imply the literals in L^+ and the agent's commitments contain the actions in C^+, and (ii) it is not possible to instantiate a literal in L^- or action in C^- such that the agent respectively believes the instantiated literal or has committed itself to the instantiated action. Somewhat simplified, a mental state condition is satisfied if the agent believes L^+ and does not believe any of the literals in L^-, and the agent is committed to C^+ and is not committed to any of the actions in C^-.

7.4.3 Executing Commitments

Since the commitments of an agent consist of the actions the agent has selected for execution, the semantics of commitments is provided by a semantics for action execution. A semantics for actions is provided relative to the meaning of the most basic or private actions. These actions define the basic capabilities of the agent and are assumed to be given. In (Shoham 1993) the meaning of private actions is not discussed in great detail. However, a number of remarks suggest that private actions should be taken as updates on the set of beliefs of the agent.[5] This is the view we will take here. For this purpose, we introduce a

[5]On p. 61 of (Shoham 1993), it is remarked that no distinction is made "between actions and facts, and the occurrence of an action will be represented by the corresponding fact

(partial) function $\mathcal{T} : \mathsf{Pact} \times \wp(\mathsf{Lit}) \to \wp(\mathsf{Lit})$ which specifies what type of update is performed by each private action.[6] The computation step resulting from executing a private action then formally is defined by the following transition rule.

Definition 7.4.3 *(private actions)*
Let a be a private action.

$$\frac{\mathcal{T}(a, \sigma) = \sigma'}{\langle\{\ldots, a, \ldots\}, \sigma\rangle \longrightarrow \langle\{\ldots\}, \sigma'\rangle}$$

A conditional action $(c_1, \ldots, c_n : a)$ is executed by testing whether there are bindings for one of the conditions c_i such that it is satisfied in the current mental state of the agent and, if the test succeeds, by committing to the action a instantiated with the computed bindings. Variables in the mental state condition thus retrieve data from the belief base and current commitments by means of pattern-matching. The values retrieved are recorded in a substitution θ and used to instantiate the action a. If the test fails, either the conditional action can be removed from the commitments or not. We have chosen to retain the conditional action as a commitment. Note that the transition rule for conditional actions is specified at the agent level (cf. chapter 3), as are all other transition rules for AGENT0. The semantics for conditional actions must be specified at the agent level because the condition of such actions must be evaluated relative to the complete current mental state of the agent. Thus, for the execution of a conditional action the context of execution, i.e. the entire mental state of the agent, is required.

Definition 7.4.4 *(conditional actions)*
Let θ be a substitution and $\Pi = \{\ldots, (c_1, \ldots, c_n : a), \ldots\}$.

$$\frac{\Pi, \sigma \models c_i\theta \text{ for some } i}{\langle\{\ldots, (c_1, \ldots, c_n : a), \ldots\}, \sigma\rangle \longrightarrow \langle\{\ldots, a\theta, \ldots\}, \sigma\rangle}$$

A refrain action δa removes actions of the form a from the set of commitments. All possible instantiations of a are removed from the current commitments. In this way, it prevents the execution of these actions. It is not clear from (Shoham 1993) if and when a refrain action itself is removed from the set of commitments. We have chosen to retain the refrain action itself after executing it, since this type of action is probably most often used for *safety reasons*. I.e., for example, to prevent destructive behaviour or to prevent certain undesirable effects of the action which is to be refrained from.[7]

holding." On p. 76 we are explained that the belief database may be updated "as a result of taking a private action".

[6]We assume that the transition function does not introduce any free variables into the belief base of an agent in compliance with our previous constraint on belief bases.

[7]We think that a refrain action makes more sense in a multi-agent setting, where requests from other agents to refrain from a particular action could be received. In the single agent setting, the conditions specifying the circumstances in which an agent should refrain from an action could probably just as well be explicitly listed in the condition part concerning the action in the list of capabilities in the program.

Definition 7.4.5 *(refrain actions)*
Let θ be a substitution.

$$\frac{\delta a,\, a\theta \in \Pi}{\langle \Pi, \sigma \rangle \longrightarrow \langle \Pi \setminus \{a\theta\}, \sigma \rangle}$$

7.4.4 Applying Commitment Rules

A commitment rule $C^+, C^- \leftharpoondown L^+, L^- \mid a$ in AGENT0 is used to make new commitments. A commitment rule does not remove old commitments, it only introduces new ones. It is possible to make a new commitment if the mental state condition $\langle L^+, L^-, C^+, C^- \rangle$ is satisfied in the current mental state.

A unique feature of AGENT0 is that *all* applicable instances of a commitment rule are fired. Thus, a new commitment is made for every set of bindings for the free variables in the mental state condition of the rule which can be derived from the current belief base and commitments of the agent.

Formally, a rule fires for each substitution that satisfies the mental state condition of the rule in the current mental state. This also is an important difference between the meaning of conditional actions and commitment rules, besides the fact that rules are never removed from the agent program. As before, we use $\mathsf{Free}(e)$ for the set of all (free) variables in e and $dom(\theta)$ for the variables in the domain of substitution θ.

Definition 7.4.6 *(rule application)*
Let Θ be the set of *all* substitutions θ such that $\Pi, \sigma \models c\theta$ and $dom(\theta) = \mathsf{Free}(L^+) \cup \mathsf{Free}(C^+)$, where $c = \langle L^+, L^-, C^+, C^- \rangle$.

$$\frac{}{\langle \Pi, \sigma \rangle \longrightarrow \langle \Pi \cup \{a\theta \mid \theta \in \Theta\}, \sigma \rangle}$$

if $C^+, C^- \leftarrow L^+, L^- \mid a$ is a commitment rules of the agent.

7.5 Decision-Making in AGENT0, AgentSpeak(L) and 3APL

One of the basic differences between a number of agent languages resides in their *control structure.* The main purpose of a control structure for an agent language is to specify which commitments, intentions or plans to deal with first and which rules to apply during the execution of an agent program. The strategy for executing commitments and applying rules in AGENT0, is to execute *all* executable commitments and apply *all* applicable rules in every cycle of the control loop of the interpreter.

The main control loop in the AGENT0 interpreter executes two consecutive phases:

1. in the first phase, *update* the commitments (the commitment rules of the agent program are used in this phase),

2. in the second phase, *execute* commitments made previously (this phase is independent from the agent program).

The update phase in the control loop again consists of two distinct steps. On the one hand, new commitments are added by firing the applicable commitment rules of the agent program, and, on the other hand, the feasibility of the agent's commitments is checked. The test to determine whether or not a commitment is feasible consists of checking a condition on the mental state to see whether or not the agent believes that it is capable of executing the commitment. In case a commitment is no longer considered feasible, the commitment is removed. The order imposed on these two steps in the AGENT0 interpreter is not discussed in (Shoham 1993).

In the execution phase, i.e. the second phase in the loop above, as many commitments as possible are executed by the AGENT0 interpreter. Since the actions that are executed are quite simple actions we suppose that each of these actions is executed completely. This remark applies in particular to conditional actions, which are executed by first performing a test and in case the test succeeds executing the action part.

In AGENT0 there are no operators for constructing complex actions. For example, sequential composition or recursive structures like procedures or plan rules are absent in AGENT0. This lack of constructs for programming control flow and abstraction is one of the things which suggests that AGENT0 supports a *bottom-up approach*. By a *bottom-up approach* we mean here that in contrast with a *top-down approach* the agent does not decide what to do next by fixing a high-level goal and computing plans, but decides what to do next by looking only at the circumstances the agent finds itself in. This lack of goal-driven behaviour of agents has been one of the reasons to design the successor language PLACA with such features in (Thomas 1993). PLACA is similar to AGENT0, but it allows planning by means of a plan library.

Another feature which also suggests AGENT0 supports a bottom-up approach is the type of rules allowed to program an agent. The rules to program agents are condition-action rules. We mean by this that the rules do not *modify* any existing (high-level) goals of the agent by substituting plans for achieving them, but just *add* new commitments to the set of commitments if the conditions of a rule are satisfied. In this sense we could say that a language like AGENT0 is *rule-driven*, while languages like AgentSpeak(L) (Rao 1996*a*) and 3APL are *goal-driven* and use a top-down approach which refines high-level goals into plans of action.

With the bottom-up and top-down approach two different styles of decision-making can be associated. These different styles of decision-making give rise to two different styles of control loops for interpreters for agent programs. The different styles of decision-making can be explained by introducing two distinct *practical syllogisms* for decision-making; one corresponding to goal-driven interpreters and one corresponding to rule-driven interpreters:

Practical Syllogism corresponding to the Goal-Driven Approach:
If (1) the agent intends to achieve a goal g, and (2) believes that g will not be achieved unless the plan p will be executed, then (Concl) the agent intends to execute plan p.[8]

Practical Syllogism corresponding to the Rule-Driven Approach:
If (1) the agent believes it is in situation S, (2) the agent already has made commitments Π concerning a set of actions, and (3) given these conditions it is advantageous to perform a, then (Concl) the agent should commit to perform action a.

An explanation of the goal-driven syllogism is to view it as a reasoning scheme which may be used by the agent to achieve a goal by means of *some* plan. The rule-driven syllogism is best explained as a reasoning scheme to guarantee commitment to *all* actions of a particular form. Thus, the first might profitably be used to infer a *possible* means to achieve a goal, while the second is more suited to be used as a means to infer the *necessity* to perform an action, i.e. to guarantee that some actions are performed in certain circumstances. The two approaches thus are dual approaches. This duality is related to the duality of the possibility and necessity operators in modal logic. For this reason, it is interesting to note that in (Shoham 1993) the concept of *obligation* is taken as basic instead of that of *motivation*. In (Shoham 1993) Shoham actually somewhat overstates, we think, the contrast between the two different modes of decision making. According to him, the decision making in AGENT0 "reflects absolutely no motivation of the agent, and merely describes the actions to which the agent is obligated." (p. 67) The tools for programming an agent in AGENT0 on the one hand, and AgentSpeak(L) and 3APL on the other hand, thus are derived from two different perspectives on decision-making.

7.6 Conclusion

By abstracting from a number of features of AGENT0, we have been able to construct an operational semantics for what we called the *single agent core of AGENT0*. This core includes beliefs, capabilities, three types of action, commitments, and commitment rules. Time and communication are not included in the single agent core. The benefits of a formal semantics in general and for AGENT0 in particular is that it provides a precise specification for an implementation of the language. The formal semantics also is an improvement over the original, informal specification presented in (Shoham 1993) and a number of gaps were identified during the construction of this semantics.

The formal semantics allows for a formal comparison, and thereby clarifies a number of differences between rule-based agent languages (cf. for a more extensive discussion Hindriks et al. (1999*b*)). The basic features of AGENT0 are very

[8]It is probably advantageous for the agent to add some conditions which state that the plan should not interfere too much with other goals of the agent. For simplicity, we have not included such conditions.

similar to those of 3APL. Capabilities correspond to basic actions, beliefs in both languages are derived from a first order language, 3APL incorporates the same type of actions except for the refrain action, and commitments in AGENT0 correspond to simple goals in 3APL. AGENT0 lacks a number of programming constructs for composing sequential, choice, and parallel goals. The main difference, however, stems from the different types of rules used in both languages. Whereas in AGENT0 it is possible to fire a rule in case a commitment is *not present*, the application of a rule in 3APL cannot be conditioned on the absence of a goal. In 3APL, however, it is possible to arbitrarily *modify* or *drop* goals, whereas in AGENT0 rules only can be used to add new commitments.

We believe that the specification of a formal semantics for AGENT0 has clarified the meaning and use of the commitment rules from AGENT0. The rules of an agent language determine to a large extent the type of decision-making associated with agents. Two types of decision-making were distinguished. In AGENT0, a bottom-up approach is used and a decision what action to do next (a new commitment) is based on the current situation the agent is in. A top-down approach is used in AgentSpeak(L) and 3APL in which a plan for action is selected to achieve a high-level goal of the agent. Accordingly, AGENT0 can be characterised as *rule-driven*, while AgentSpeak(L) and 3APL can be characterised as *goal-driven*.

AGENT0 has a number of obvious limitations. For this reason, a successor language called PLACA has been designed (Thomas 1993). In particular, the language PLACA extends AGENT0 agents with planning capabilities similar to plan rules in 3APL. The approach that is used for PLACA is very similar to that of AGENT0. Again, a modal logic is designed to define a number of agent concepts and a programming language with similar, but not quite the same, concepts is proposed. The set of concepts incorporated into PLACA is larger than that of AGENT0 and therefore the language is also more complicated. In fact, one could argue that the simplicity of agent-oriented programming that is obtained by using a minimal set of agent concepts is lost in PLACA. Moreover, since PLACA is quite similar to AGENT0 and its most prominent contribution is the addition of planning capabilities which are also present in 3APL, we do not discuss PLACA in detail here.

CHAPTER 8

Expressive Power of Agent Languages

The transition style semantics that we have been using throughout this thesis provides a basis for a formal comparison of agent languages. The operational semantics associated with an agent defines the potential behaviour of that agent. The semantics thus allows for a comparison of the *behaviour* of two agents. Formally, the potential behaviour of an agent is defined by a *set of computations.*

Presented with a formal definition of the behaviour of agents by means of computations, we need to consider when the behaviour of two agents is identified. We may not want to take all aspects of the computational behaviour into account and abstract from a number of features. It is thus important to specify which aspects of the agent's behaviour are considered, which naturally leads to the concept of *observation.*

By means of the concepts of a computation and that of an observation a formal definition of two agents that simulate each other is provided. An agent language defines a set of possible agents $\mathcal{A}$. The comparison of two agent languages thus involves a comparison of their respective sets of agents. The basic idea is that the comparison of one set of agents $\mathcal{A}$ with another set of agents $\mathcal{B}$ consists of two steps: (i) for each agent $A \in \mathcal{A}$ find a corresponding agent $B \in \mathcal{B}$, and (ii) show that the behaviour of agent A can be simulated by the behaviour of agent B. Because in principle any Turing-complete language can simulate another Turing-complete programming language, we need to formulate some additional requirements to use a simulation approach for comparing the *expressive power* of two agent languages. Intuitively, agent B should be structure-preserving and generate the same type of behaviour in a similar way.

8.1 Computations and Observables

Agents or agent programs are the entities of a programming language that are executed, and that give rise to computations. A transition style semantics defines a *transition relation* $\longrightarrow$ which specifies the *computation steps* that an agent can perform. A transition system thus defines in a natural way the set of computations that are associated with an agent.

Definition 8.1.1 *(computation)*
A *computation* of an agent A is a finite or infinite sequence

$$A_0, A_1, \ldots, A_i, A_{i+1}, \ldots$$

such that $A_0 = A$ and for all i: $A_i \longrightarrow A_{i+1}$. A transition $A \longrightarrow A'$ is called a *computation step* (of agent A).

The set of all computations associated with agent A is called the *operational meaning* of agent A. This set specifies what an agent can do, or, in other words, its behaviour. A transition relation may be *labelled* to distinguish different types of computation steps. For example, a transition due to the execution of an action a may be labelled with this action, which is written as $A \xrightarrow{a} A'$.

Given the above definition of a computation we can define various notions of *observables*. For example, we may want to observe the sequence of belief bases extracted from a computation, or the sequence of basic actions corresponding to single transitions in a computation, for example for planning (cf. Giacomo et al. (1997)). Belief bases and actions are different types of observables. A belief base is part of the *internal state* of the agent, whereas an action that is observed to have happened reveals the *type of computation step* that has been performed. Belief bases are extracted from the agents in a computation and are *state-based observables*. Actions or other types of computation steps that are distinguished are extracted from computation steps and are *action-based observables*.

We will assume that action-based observables are always given by means of labels that are associated with transitions. To extract state-based observables from a computation, we will assume that an *observation function* $\mathcal{O} : \mathcal{A} \to \Omega$ - where $\mathcal{A}$ is a set of agents and Ω is a set of state-based observables - has been given. $\mathcal{O}$ defines which parts of the state of an agent are visible to an external observer. Note in particular that the function $\mathcal{O}$ allows an observer to *observe changes* in the agent's state during a computation.

8.2 Bisimulating Agents

The comparison of two agents is based on a notion of *simulation*. Intuitively, two agents can simulate each other if they can generate the same behaviour. Simulation is defined in terms of computations and the concept of observation. An agent A is able to simulate a second agent B if every computation of agent B matches with a computation of agent A, and vice versa. To be somewhat more precise, two agents A and B are similar if for every computation of agent A there exists a matching computation of agent B, and vice versa. For two computations to match, the individual computation steps of the computations are compared. The basic idea is that two agents that can simulate each other should be able at each moment during a computation to generate a similar computation step. Computation steps are similar if the observables of the steps are identical. The idea to identify agents if they are able to simulate each other's computation steps is called *(strong) bisimulation* in the literature (cf. Park (1980) and Milner (1989)). A bisimulation is a binary relation between agents.

Definition 8.2.1 *(strong bisimulation)*
Let $\mathcal{A}$ and $\mathcal{B}$ be two sets of agents. A binary relation $R \subseteq \mathcal{A} \times \mathcal{B}$ over agents is a *strong bisimulation* if $(A, B) \in R$ implies that,

(i) Whenever $A \xrightarrow{l} A'$ then, for some B', $B \xrightarrow{l} B'$, and $(A', B') \in R$,

(ii) Whenever $B \xrightarrow{l} B'$ then, for some A', $A \xrightarrow{l} A'$, and $(A', B') \in R$, and

(iii) $\mathcal{O}(A) = \mathcal{O}(B)$.

Observe that a bisimulation is an equivalence relation on agents. The criteria for identifying two agents are derived from the observation function $\mathcal{O}$ (which defines the state-based observables) and the type of computation steps that are distinguished by labels (which define the action-based observables). By either mapping all agents onto a single value (i.e. $\mathcal{O}(A) = \perp$ for all agents A) or by not distinguishing computation steps by means of labels the state-based and action-based observables can be trivialised. However, even if both state-based and action-based observables are trivial, a bisimulation still requires a potential for behaviour. That is, if agents A and B bisimulate, each of them must be able to perform a computation step if the other is able to do so.

The concept of bisimulation is the basic tool that we will use for comparing agents and agent languages. The notion of strong bisimulation, however, imposes a very strong condition for the identification of two agents. A relation on agents is a strong bisimulation if and only if each computation step of an agent A that is bisimular to an agent B is matched by a computation step of agent B, and vice versa. This may be asking too much if we want to compare different agent languages because it leaves very little room for differences.

From the point of view of comparing programming languages, it may not be relevant to distinguish all the peculiar details of one language with those of the other. Some mechanisms present in a language may be considered 'implementation details' from the perspective of another language. For example, the use of a *stack* of plans in AgentSpeak(L) from the perspective of 3APL provides too much detail, whereas in the semantics of ConGolog the details of variable renaming which are present in the semantics of 3APL are hidden.

To relax the conditions imposed by strong bisimulation, we might argue that some computation steps $A \longrightarrow A'$ are not observable. Several criteria may be used to specify which computation steps are not observable. However, there is one type of computation step that in particular comes to mind, namely computation steps $A \longrightarrow A'$ such that $\mathcal{O}(A) = \mathcal{O}(A')$. The idea is that computation steps which cannot be distinguished by an external observer need not be simulated. From the point of view of the simulation, these steps can be considered 'implementation details' that are hidden from the observer. We call such 'hidden' steps *silent steps.*

The idea is to allow agents to perform silent steps which are not necessarily matched by computation steps of the other agent. As a consequence, an observable computation step of one agent may be matched by another agent by performing zero or more silent steps possibly combined with a single non-silent step of the other agent. The concept of simulation that we obtain in this way by relaxing the requirement that every computation step needs to be simulated is called *weak bisimulation.*

To formalise this idea, a special label i is introduced. A transition is singled out as a silent step in case the label i is associated with the transition and considered a non-silent step otherwise. (For the moment, we assume that no other labels are used.) A new transition relation $\Rightarrow$ is derived from the original transition relation $\longrightarrow$ that abstracts from such silent steps. A step $A \Rightarrow A'$ may involve a number of silent steps and possibly a non-silent step.

Definition 8.2.2 *(abstracting from silent steps)*
Let $\longrightarrow$ be a transition relation on agents from $\mathcal{A}$, i be a label that is associated with a transition to mark it as a silent step, and $\longrightarrow^*$ denote the transitive closure of a relation $\longrightarrow$. Then the *transition relation* $\Rightarrow$ is defined by:

$A \Rightarrow A'$ iff there are agents $A_1, A_2 \in \mathcal{A}$ such that:
$A \xrightarrow{i}^* A_1, A_1 \longrightarrow A_2$ *or* $A_1 = A_2$, and $A_2 \xrightarrow{i}^* A'$

Observe that $\xrightarrow{i}^* \subseteq \Rightarrow$. The notion of weak bisimulation is based on the idea that a single computation step of one agent may be simulated by the derived step relation $\Rightarrow$. That is, a single computation step $A \longrightarrow A'$ may be matched by an agent B by performing a step $B \Rightarrow B'$. The step $B \Rightarrow B'$ may involve multiple silent steps as well as one non-silent step.

Definition 8.2.3 *(weak bisimulation)*
Let $\mathcal{A}$ and $\mathcal{B}$ be two sets of agents. A binary relation $R \subseteq \mathcal{A} \times \mathcal{B}$ over agents is a *weak bisimulation* if $(A, B) \in R$ implies,

(i) Whenever $A \longrightarrow A'$ then, for some B', $B \Rightarrow B'$, and $(A', B') \in R$,

(ii) Whenever $B \longrightarrow B'$ then, for some A', $A \Rightarrow A'$, and $(A', B') \in R$, and

(iii) $\mathcal{O}(A) = \mathcal{O}(B)$.

The definition 8.2.1 of strong bisimulation and the definition 8.2.3 of weak bisimulation look very similar. In fact, every strong bisimulation is a weak bisimulation. A weak bisimulation, however, is not always a strong bisimulation because of the possibility that silent steps may have been used to simulate non-silent steps. Note that in case $\mathcal{O}(A) = \mathcal{O}(A')$ is used as a criteria to single out computation steps $A \longrightarrow A'$ as silent steps, a non-silent step of one agent can only be simulated by another agent if it performs at least one non-silent step itself.

It is not difficult to extend the definition of weak bisimulation and also allow other labels than i to be associated with transitions. The extension involves associating labels with the step relation $\Rightarrow$ from definition 8.2.2. We stipulate that a label i is associated with a step $A \Rightarrow A'$ if the step only involves silent steps (or no steps at all), written as $A \overset{i}{\Rightarrow} A'$; if a step $A \Rightarrow A'$ involves a non-silent step labelled l, then this label is associated with the step, written as $A \overset{l}{\Rightarrow} A'$. The changes to definition 8.2.3 then are straightforward.

8.3 Translation Bisimulation

So far, we have been discussing the comparison of two sets of agents, and have paid little attention to the programming languages used to construct these agents. We are interested in comparing different agent languages and features of these languages, however. In particular, we would like to study the *expressive power* of agent languages. For this purpose, it is not good enough to construct a bisimulation for *some* set of agents. To compare two languages, we need to systematically associate agents from one language with that of another. That is, if an agent language defines a set of agents $\mathcal{A}$, for each of these agents we need to find a corresponding agent in the other language. The concept of a *translation bisimulation* is developed here to formalise this idea.

Our aim is to be able to compare the expressive power of one language with that of another. Informally, this means that we are interested in finding out whether or not every agent that can be programmed in a given language can be translated into an agent of the second language that 'can do the same things'. The first language is called the *source language*, and the second language in which those of the first are translated is called the *target language*. In the sequel, we assume that the source language defines a set of agents $\mathcal{A}$ and the target language defines a set of agents $\mathcal{B}$.

For our purposes, it is thus important to require that *every* agent from the source language is *translated* into an agent of the target language. A method for mapping agents from $\mathcal{A}$ to agents from $\mathcal{B}$ is required to translate between the source and the target language. Such a method systematically associates an agent from $\mathcal{B}$ with every agent from $\mathcal{A}$ and is called a *translation function.* A translation function τ may define a weak bisimulation R, i.e. $\tau = R$. In that case, we obtain a *special* case of weak bisimulation that is also called a *p-morphism* in the literature (cf. Segerberg (1970)). A p-morphism is a (weak) bisimulation such that the bisimulation relation is a function. We will call a translation function that is a p-morphism a *translation bisimulation.*

Because we allow the source and the target language to be different languages, we need to take into account a number of differences. First, we need a translation function which translates agent programs from the source to the target language. Secondly, we must keep in mind that the sets of observables may be defined differently for the source and the target language. To compensate for this second difference, we introduce a mapping that we call a *decoder*. A decoder maps observables from the target language back onto observables of the source language. A decoder thus is a function $\delta : \Omega_{\mathcal{B}} \rightarrow \Omega_{\mathcal{A}}$ where $\Omega_{\mathcal{A}}$ and $\Omega_{\mathcal{B}}$ are the observables from the source and target language. In case we need to decode labels from the target language, a decoder may also be extended to action-based observables. The reason that a decoder decodes observables *from the target language into the source language* is that we assume that the target language is used to simulate the source language, but the source language does not have to simulate the target language.

Definition 8.3.1 *(translation bisimulation)*
Let the following be given:

- $\longrightarrow_{\mathcal{A}}$, $\longrightarrow_{\mathcal{B}}$ are two transition relations defined on the sets of agents $\mathcal{A}$ and $\mathcal{B}$, respectively,
- $\mathcal{O}_{\mathcal{A}} : \mathcal{A} \rightarrow \Omega_{\mathcal{A}}$ and $\mathcal{O}_{\mathcal{B}} : \mathcal{B} \rightarrow \Omega_{\mathcal{B}}$ are two functions that define the sets of observables associated with agents from $\mathcal{A}$ and $\mathcal{B}$, respectively,
- $\tau : \mathcal{A} \rightarrow \mathcal{B}$ is a total mapping from $\mathcal{A}$ to $\mathcal{B}$,
- $\delta : \Omega_{\mathcal{B}} \rightarrow \Omega_{\mathcal{A}}$ is a *decoder*,
- if it exists, the distinction between silent and non-silent steps is clearly formulated for $\longrightarrow_{\mathcal{A}}$ and $\longrightarrow_{\mathcal{B}}$.

Then we have that τ is a *translation bisimulation* if, for every $A \in \mathcal{A}$, $B = \tau(A)$ implies,

- Whenever $A \longrightarrow_{\mathcal{A}} A'$, then $B \Rightarrow_{\mathcal{B}} B'$, such that $B' = \tau(A')$,
- Whenever $B \longrightarrow_{\mathcal{B}} B'$, then for some A', $A \Rightarrow_{\mathcal{A}} A'$, such that $B' = \tau(A')$, and
- $\delta(\mathcal{O}_{\mathcal{B}}(B)) = \mathcal{O}_{\mathcal{A}}(A)$.

The concept of a translation bisimulation is illustrated in figure 8.1. $\mathcal{L}$ denotes the source language and $\mathcal{L}'$ denotes the target language in the figure. Note that, although this is not depicted in the figure, the agents may perform more than one step to simulate a step of the other agent on condition that these extra steps are silent steps. In the figure, this means that the arrows are used both to depict a $\longrightarrow$ as well as a $\Rightarrow$ step. A translation function τ maps an agent $A \in \mathcal{A}$ from the source language $\mathcal{L}$ to an agent $\tau(A) \in \mathcal{B}$ from the target language $\mathcal{L}'$. For each agent $A \in \mathcal{A}$ a computation step $A \longrightarrow_{\mathcal{A}} A'$

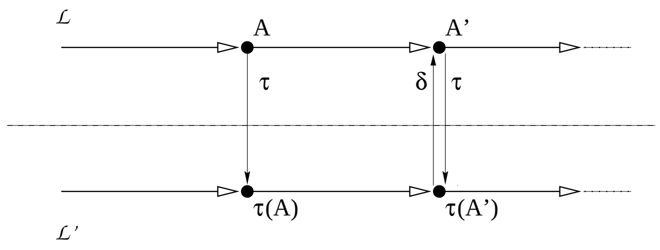

Figure 8.1: Translation Bisimulation

must be matched by a corresponding computation $\tau(A) \Rightarrow_{\mathcal{B}} \tau(A')$. This step should have the same observable effects. And vice versa, a step of $\tau(A)$ must be matched by agent A.

The first condition in definition 8.3.1 states that a computation step of the source agent A is matched by a corresponding computation from the target agent $\tau(A)$. The third condition requires that it is possible to extract the same observable information from the target agent as from the source agent. This condition excludes a trivial choice of observables $\Omega_{\mathcal{B}}$ for the target language, i.e. $\Omega_{\mathcal{B}} = \{\perp\}$. Together, these conditions guarantee that the behaviour produced by the source agent A can be simulated by the target agent $\tau(A)$.

The source agent, however, must also be able to simulate the behaviour of the target agent. Otherwise, the two agents cannot be considered equivalent. In other words, we also have to prove that all possible behaviour of the target agent $\tau(A)$ can be accounted for on the basis of the behaviour of agent A. Agent $\tau(A)$ should generate the same behaviour as agent A, but is not allowed to generate *alternative* computations. If such extra computations would be allowed, a target agent for simulating a source agent could be selected that generates as much computations as possible, which would trivialise the simulation result. Formally, these requirements are captured by the second condition in definition 8.3.1. In short, it states that every computation step of the target agent $\tau(A)$ also must be simulated by the source agent A.

A translation bisimulation offers two features for dealing with differences between the source and the target language. First, a translation function may translate constructs and features of the source language into the constructs and features of the target language. And secondly, a decoder may be used to decode observables of the target language back into those of the source language. There are no facilities introduced yet to deal with different syntactic mechanisms that transform an agent during a computation. (A translation function maps a source agent onto a 'canonical' target agent, and cannot be used to compensate for such mechanisms.) For example, a language may use a renaming scheme for variables that is not present in another language (cf. chapter 10 which compares 3APL with ConGolog). In general, such syntactic mechanisms give rise to slightly different agents. From the point of view of an external observer, however, these agents cannot be distinguished and one would like to abstract from these minor syntactic differences.

Traditionally, entities that are identical except for minor syntactic differences, have been identified by means of a process called α-conversion. It is this notion that we will use here too. The idea is to allow an agent to perform so-called α-conversion steps to simulate a step of another agent. The point is that a translation function may map a source agent onto a canonical target agent that might be slightly different with respect to minor syntactic features from the target agent that is needed in the simulation step. To compensate for such differences, then, α-conversion steps may be performed during the simulation.

An α-conversion step is written like $A \xrightarrow{\alpha} A'$. α-conversion steps need not, however, be computation steps defined by the transition relation $\longrightarrow$. For the purpose of constructing a translation bisimulation, these steps may be added. Of course, we cannot arbitrarily introduce α-conversion steps. Whether or not a step may be used as an α-conversion step depends on the context. In particular, it depends on the observation function $\mathcal{O}$ that is used. Minimally, an α-conversion step $A \xrightarrow{\alpha} A'$should satisfy the following condition: A and A' are bisimilar with respect to the observation function $\mathcal{O}$, where $\mathcal{O}$ is given by the context of the translation bisimulation that is constructed. Thus apart from silent steps, in the context of a translation bisimulation we also allow that α-conversion steps are performed in an $\Rightarrow$-step.

8.4 Expressive Power

The concept of a translation bisimulation is particularly useful for comparing the expressive power of agent programming languages. Translation bisimulation specifies a number of conditions that must be satisfied for one language to have at least the same expressive power as another language. To establish that a language $\mathcal{L}$ can express everything that another language $\mathcal{L}'$ can express, we must construct a translation bisimulation that relates *all* agents in the source language $\mathcal{L}'$ to *some* suitable set of agents from the target language $\mathcal{L}'$. This shows that the target language can simulate arbitrary agents from the source language.

However, the notion of translation bisimulation is still not strong enough to *define* expressive power, since programming languages that are Turing-complete can simulate any other Turing-complete programming language. The ingredient that is still missing from the definition of translation bisimulation is the notion of *conceptual structure*. Different programming languages incorporate different concepts. In fact, this is the sole reason for the existence of so many different languages: A programming language may not offer more computational power, but it may provide a different set of tools (concepts) to solve a programming problem. Intuitively, a translation function τ thus should also preserve the *global structure of an agent* if we want to compare the expressive power of programming languages (cf. Felleisen (1990)).

It is not so easy to formalise this intuitive constraint on a translation bisimulation. The formalisation of the constraint depends on the definition of agents in the source and target language. If we assume that both languages are defined inductively, the preservation of structure can be captured by the requirement that both the translation function τ and the decoder δ are *compositional.*

Somewhat more formally, we will call a translation compositional if every operator op of the source language is translated into a *context* $C[x_1, \ldots, x_n]$ (where n is the arity of op) of the target language such that an expression $op(e_1, \ldots, e_n)$ is translated into $C[\tau(e_1), \ldots, \tau(e_n)]$. In order to account for the complex structure of an agent (that is, its various components comprising its beliefs, goals, intentions, etc.) we will assume that an agent is a tuple $A = \langle E_1, \ldots, E_n \rangle$, where each of the E_i is a subset of the expressions of a programming language. A mapping from agents $\langle E_1, \ldots, E_n \rangle$ written in the source language to agents $\langle F_1, \ldots, F_m \rangle$ in the target language ($n \neq m$ is allowed) is then induced by the translation function τ and a *pre-specified selection criterion* that determines for each expression $e \in E_i$ a corresponding target component F_j such that $\tau(e) \in F_j$. Similarly, compositionality of δ can be formalised.

Definition 8.4.1 *(embedding $\mathcal{A}$ into $\mathcal{B}$)*
Let $\mathcal{O}_\mathcal{A} : \mathcal{A} \to \Omega_\mathcal{A}$ and $\mathcal{O}_\mathcal{B} : \mathcal{B} \to \Omega_\mathcal{B}$ be two observation functions for the set of agents $\mathcal{A}$ from $\mathcal{L}$ and the set of agents $\mathcal{B}$ from $\mathcal{L}'$.
Then we say that $\mathcal{L}'$ *has at least the same expressive power* as $\mathcal{L}$ if there is a mapping $\tau : \mathcal{A} \to \mathcal{B}$ and a mapping $\delta : \Omega_\mathcal{B} \to \Omega_\mathcal{A}$ which satisfy the following conditions:

E1 τ and δ are compositional,

E2 τ is a translation bisimulation.

If a language $\mathcal{L}'$ has at least the same expressive power as another language $\mathcal{L}$, we also write $\mathcal{L} \leq \mathcal{L}'$.

Corollary 8.4.2 *(expressive power is a transitive relation)*
Let $\mathcal{L}, \mathcal{L}', \mathcal{L}''$ be programming languages.
Then we have that if $\mathcal{L} \leq \mathcal{L}'$ and $\mathcal{L}' \leq \mathcal{L}''$, then also $\mathcal{L} \leq \mathcal{L}''$. That is, $\leq$ is transitive.

Proof: Informally, the corollary states that if $\mathcal{L}'$ has at least the expressive power of $\mathcal{L}$, and $\mathcal{L}''$ has at least the expressive power of $\mathcal{L}'$, then $\mathcal{L}''$ has at least the expressive power of $\mathcal{L}$. The result follows from the fact that the composition of two compositional translation functions again is compositional and the fact that composing two translation bisimulations again yields a translation bisimulation. $\Box$

Remark 8.4.3 *(interacting agent components)*
The translation of the different components E_i that make up an agent $A = \langle E_1, \ldots, E_n \rangle$ is rather subtle. The most simple construction, of course, is to define a translation function τ that maps the components E_i of agent A to

different components of the target agent $\tau(A)$. Such translation schemes still would allow for mapping expressions from two components E_i and E_j $(i \neq j)$ to a single target component $\tau(E_i) \cup \tau(E_j)$, or for mapping expressions from a single component E_i to different target components.

Simple translation schemes such as those of the previous paragraph, however, cannot account for possible semantic interactions between agent components. This would be the case if the meaning of an expression in one agent component may be altered by the presence of an expression in another component. This phenomenon may arise in 3APL agents because of the presence of practical reasoning rules. For example, the presence of the swap rule $X;\ Y \leftarrow Y;\ X$ (cf. chapter 4) changes the meaning of sequential composition.

To be able to deal with such shifts in meaning, it should be possible that the translation of expressions in one agent component into another language depend on the contents of a second component. It thus should be possible to parameterise the translation function τ with components. That is, the translation of a component E_i by τ may be dependent on another component E_j, which can be expressed by $\tau(E_j)(E_i)$.

For example, a more complex translation scheme allows the translation of a 3APL agent $\langle \sigma, \{\mathsf{a};\ \mathsf{b}\}, \{X;\ Y \leftarrow Y;\ X\}\rangle$ into another 3APL agent $\langle \tau(\sigma), \tau(\{X;\ Y \leftarrow Y;\ X\})(\{\mathsf{a};\ \mathsf{b}\}), \tau(\{X;\ Y \leftarrow Y;\ X\})\rangle = \langle \sigma, \{\mathsf{a};\ \mathsf{b} + \mathsf{b};\ \mathsf{a}\}, \varnothing\rangle$.

A particularly interesting issue in this context concerns the expressive power of individual constructs in a single language. More specifically, it is interesting to know whether or not a programming construct adds expressive power to a language. In case it does not, the programming construct might be considered redundant and could be eliminated from the language. For example, it is well-known that in imperative programming the **repeat** ... **until** ... construct can be eliminated if both the **while** ... **do** ... construct and sequential composition are available in the language.

The fact that a programming construct can be eliminated from a language means that a subset of that language has at least the same expressive power. The formal definition of eliminability of an operator *op* is given next.

Definition 8.4.4 *(eliminability)*
Let $\mathcal{O}_\mathcal{A} : \mathcal{A} \to \Omega_\mathcal{A}$ and $\mathcal{O}_\mathcal{B} : \mathcal{B} \to \Omega_\mathcal{B}$ be two observation functions for the set of agents $\mathcal{A}$ and the set of agents $\mathcal{B}$. We assume that agents from $\mathcal{A}$ and $\mathcal{B}$ are written in the same language $\mathcal{L}$.
Then we say that a programming operator *op* in the language $\mathcal{L}$ is *eliminable* if there is a mapping $\tau : \mathcal{A} \to \mathcal{B}$ and a mapping $\delta : \Omega_\mathcal{B} \to \Omega_\mathcal{A}$ which satisfy the following conditions:

F1 τ and δ are compositional,

F2 τ is a translation bisimulation, and

F3 $\mathcal{B} \subset \mathcal{A}$, such that the operator *op* does not occur in any of the agents in $\mathcal{B}$.

The definition 8.4.4 of eliminability is very similar to that of definition 8.4.1. The two first conditions **F1** and **F2** are identical to **E1** and **E2** of definition 8.4.1. The only difference is that the target language is required to be a subset of the source language by condition **F3**. Although eliminability requires the syntax of the target language to be a subset of the source language, definition 8.4.4 allows that there are small differences in the meaning of the same (syntactic) operators in the source and target language. An illustration of these changes in meaning between source and target language will be presented in the next chapter where we show that events are eliminable from AgentSpeak(L).

8.5 Conclusion

In this chapter, we developed a method for the comparison of agent languages. In particular, the method supports a comparison of the relative expressive power of two languages. The method is based on the comparison of the (observable) behaviour of agents. The behaviour of an agent is supposed to be formalised by means of an operational semantics that defines the possible computations that an agent can engage in. Based on the concepts of a computation and that of an observation, several notions of *bisimulation* were introduced.

The most important concept developed in this chapter is that of a *translation bisimulation*. A translation bisimulation systematically associates agents from a so-called target language with every agent from a source language. The agents from the source and target language that are related by a translation bisimulation are required to generate the same behaviour.

For the comparison of expressive power, translation functions are required to preserve the global structure of agents. Formally, a translation function should be *compositional*. The formal definition of the relative expressive power of two programming languages is used in the next two chapters to compare AgentSpeak(L) and ConGolog with 3APL.

CHAPTER 9

An Embedding of AgentSpeak(L) in 3APL

The language AgentSpeak(L) (Rao 1996*a*) is similar in many respects to 3APL. Nevertheless, there are a number of conceptual differences between both agent languages. It is therefore interesting to compare these languages to obtain a better insight into what type of agents both languages define and into the terminology that is used to define both languages. In particular, it is useful to formally compare both languages and provide a detailed comparison that allows us to evaluate the differences in a rigorous way.

To this end, we use the bisimulation technique that has been studied in the previous chapter. We construct a formal embedding of the agent language AgentSpeak(L) in 3APL. This shows that 3APL has at least the same expressive power as AgentSpeak(L). The comparison yields some new insights into the operation of the agents in both languages and reveals some formal relations between concepts used in 3APL and those used in AgentSpeak(L). Moreover, the results proven in this chapter show that it is possible to substantially simplify AgentSpeak(L).

9.1 Overview of AgentSpeak(L)

We first provide a short and informal overview of AgentSpeak(L) and then formally introduce the language. The languages AgentSpeak(L) and 3APL share many features. One of the more important of these shared features is that both languages are *rule-based*. In AgentSpeak(L) the rules embody the know-how of the agent, and are used for planning. The informal reading of these rules is: If an agent wants to achieve goal G and believes that situation S is the case, then plan P might be the right thing to do. Recall that the same type of rules for planning is available in 3APL (cf. chapter 2).

An agent of the programming language AgentSpeak(L) consists of *beliefs*, *goals*, *plan rules*, *intentions*, and *events* which make up its mental state. A set of *actions* for changing its environment is associated with an agent. Actions are executed as a result of executing adopted plans. Adopted plans are called *intentions* in AgentSpeak(L).

The *beliefs* of an AgentSpeak(L) agent represent the situation the agent thinks it is in and are similarly defined as those of 3APL and AGENT0. In AgentSpeak(L) two types of goals are distinguished: *achievement goals* and *test goals*. Both types of goals are similar to those of 3APL. A test goal can be used to inspect the beliefs of the agent. An achievement goal is a state of affairs desired by the agent. The means to achieve such a state are provided by *plan rules*. An attempt to accomplish an achievement goal is initiated by a search in the plan library for a plan rule that provides the appropriate means to succeed. If an appropriate plan rule is available, the agent may select the plan and start executing the plan as specified by the rule.

Plans are hierarchically structured. A plan may - among other things - consist of achievement goals which represent a high-level goal that needs to be achieved to execute the plan. To achieve these achievement goals, an agent needs to find appropriate plans for these goals in turn. A plan that is adopted to achieve an achievement goal is put on a stack of plans. Such a stack of plans is called an *intention* in AgentSpeak(L). The structure of an intention (i.e. a stack of plans) is related to the occurrences of achievement goals in the plans of which the intention consists. Each entry of a stack of plans is adopted as a plan to achieve the first occurrence of an achievement goal in the plan one entry lower in the stack, except, of course, for the bottom element.

An agent acts on the basis of its intentions. An agent first executes the plan that is last pushed onto an intention. In case it finishes executing a plan on the stack, execution continues with the next plan on the stack. A plan may specify that the agent should perform an action or that the agent should accomplish a test or achievement goal.

9.2 Outline of the Proof

The claim that 3APL has at least the same expressive power as AgentSpeak(L) is proven in two steps. Each of these steps deals with a conceptual difference between AgentSpeak(L) and 3APL. The main conceptual differences between AgentSpeak(L) and 3APL are the concepts of an *event* and an *intention* which do not have obvious counterparts in 3APL.

In the first step (section 9.3), we show that the concept of an event can be eliminated from the language AgentSpeak(L) without reducing its expressive power. To prove this, we first define a new language that is called AgentSpeak(1). AgentSpeak(1) is a subset of the language AgentSpeak(L) and does not include events. We show that AgentSpeak(1) has at least the same expressive power as AgentSpeak(L). Because AgentSpeak(1) is a proper subset of AgentSpeak(L) we then can conclude that events in AgentSpeak(L) are eliminable in the formal sense of definition 8.4.4.

In the second step (section 9.4), we show that the concept of an intention can be simulated by 3APL goals. For this purpose, we define a language called AgentSpeak(2) and show that this language has at least the same expressive power as AgentSpeak(1). AgentSpeak(2) is a proper subset of 3APL. 3APL thus has at least the expressive power of AgentSpeak(2). Since AgentSpeak(2) has at least the expressive power of AgentSpeak(1) and AgentSpeak(1) has the expressive power of AgentSpeak(L), by transitivity it then follows that 3APL has at least the same expressive power as AgentSpeak(L).

9.3 The Eliminability of Events

In this section, we show that events can be eliminated from AgentSpeak(L). This is the first step in the proof that AgentSpeak(L) agents can be simulated by 3APL agents. A compositional translation function τ_1 that translates AgentSpeak(L) agents to a language called AgentSpeak(1) is constructed. Then a proof is presented that shows that the function τ_1 defines a translation bisimulation. It follows that AgentSpeak(1) has at least the same expressive power as AgentSpeak(L) and, since AgentSpeak(1) is a subset of AgentSpeak(L) which does not include events, that events are eliminable from AgentSpeak(L).

First, we introduce both the syntax and the semantics of the programming languages AgentSpeak(L) and AgentSpeak(1), respectively. The definition of AgentSpeak(L) is based on the description of AgentSpeak(L) in (Rao 1996*a*). The syntax of AgentSpeak(L) is copied without changes from (Rao 1996*a*). The semantics, however, has been changed to correct for certain omissions. The second language AgentSpeak(1) is a proper subset of AgentSpeak(L). One of the main differences between AgentSpeak(L) and AgentSpeak(1) consists in the presentation of the operational semantics. Whereas the semantics of AgentSpeak(L) is defined by a so-called proof system which is introduced below, the semantics of AgentSpeak(1) is defined by a transition system. Transition systems were also used to define the semantics of 3APL. Both systems define the operational semantics of agents and a step relation that specifies what computation steps an agent can perform. Based upon these two semantics, an embedding of AgentSpeak(L) in AgentSpeak(1) is constructed.

9.3.1 The Syntax of AgentSpeak(L)

The beliefs and actions of AgentSpeak(L) agents are defined completely analogous to those of AGENT0 agents. AgentSpeak(L) beliefs are identified with the set of literals of a first order language. AgentSpeak(L) beliefs are the same as AGENT0 beliefs and therefore we refer to definition 7.3.2 that defines terms, atoms, and literals for AGENT0 for a precise definition of these same notions for AgentSpeak(L). The terminology in (Rao 1996*a*) is slightly different from ours. A ground atom is called a *base belief* and the set of literals are called *belief literals*. Similarly, the set of (private) actions as defined in 7.3.3 for AGENT0 is the same as that for AgentSpeak(L). This set is denoted by Act here. Actions are the basic means for an agent to achieve its goals and any specific set associated with an agent defines the capabilities of that agent.

Conceptually, a difference is made between *goals* and *intentions*. The notion of a goal in AgentSpeak(L) therefore is somewhat different from the goal concept in 3APL. In AgentSpeak(L), two different types of goals are distinguished. An *achievement goal* $!\phi$ denotes the fact that an agent has a goal to establish a state of affairs where ϕ is the case. An achievement goal can be part of a larger plan. The second type of goal is a *test goal* $?\phi$ which is used to inspect the belief base of the agent. Apart from syntactic differences both types of goals correspond to the same notions in 3APL.

Definition 9.3.1 *(test and achievement goals)*
The set of AgentSpeak(L) *goals* Agoal is defined by:

- If $\phi \in \mathsf{At}$, then $!\phi \in \mathsf{Agoal}$, called *achievement goals*,
- If $\phi \in \mathsf{At}$, then $?\phi \in \mathsf{Agoal}$, called *test goals*.

In (Rao 1996*a*), four types of so-called *triggering events* are introduced. Triggering events are used by AgentSpeak(L) agents to associate plans with achievement goals and to respond to changes in the agent's environment. Triggering events have no analogues in 3APL. The idea is that a triggering event $+!\phi$ deals with the addition of an achievement goal $!\phi$ and a triggering event $-!\phi$ with the deletion of $!\phi$. Similarly, $+?\phi$ deals with the addition of a belief ϕ and $-?\phi$ with the removal of such a belief. The formal semantics of triggering events, however, is only specified for the addition of an achievement goal in (Rao 1996*a*). The meaning of the other triggering events is less clear, and therefore we do not consider these in the present chapter. A triggering event $+!\phi$ is generated when a plan for an achievement goal $!\phi$ has to be found. A triggering event thus signals the need for an appropriate plan for $!\phi$.

Definition 9.3.2 *(triggering events)*
The set of *triggering events* TrigEv is defined by:

- If $!\phi \in \mathsf{Agoal}$, then $+!\phi \in \mathsf{TrigEv}$.

Plan rules in AgentSpeak(L) provide the means for achieving an achievement goal and are similar to the same rules in 3APL. The associated plans in these rules are recipes that code the know-how of the agent. Again, there are some differences in syntax and terminology with 3APL. A plan rule is of the form $e : b_1 \wedge \ldots \wedge b_n \leftarrow h_1; \ \ldots; \ h_n$. e must be a triggering event and is called the *head* of the plan rule. It indicates for which achievement goal the plan rule provides a plan. The sequence $h_1; \ \ldots; \ h_n$ specifies the *plan* associated with the rule and is also called the *body* of the rule. Finally, $b_1 \wedge \ldots \wedge b_n$ is a condition that specifies in which contexts the plan may be considered. This condition is called the *context* of the plan rule. Empty bodies are allowed in (Rao 1996*a*), but for simplicity we do not consider them here.

Definition 9.3.3 *(plan rules)*
The set of *plan rules* $\mathsf{PlanRule}$ is defined by:

- If $e \in \mathsf{TrigEv}$, $b_1, \ldots, b_m$ are belief literals, and $h_1, \ldots, h_n \in (\mathsf{Agoal} \cup \mathsf{Act})$, then $e : b_1 \wedge \ldots \wedge b_n \leftarrow h_1;\ \ldots;\ h_n \in \mathsf{PlanRule}$.

The adoption of plans by an AgentSpeak(L) agent to achieve its goals results in *intentions.* AgentSpeak(L) intentions are most similar to 3APL goals. An intention is a stack of plans. They are used to record which plan was adopted by an agent to achieve a specific achievement goal. The idea is that a plan in an intention stack provides the means for an achievement goal that occurs in a plan one entry below it in the stack (if there is such an entry). Intentions are executed by agents by executing the actions and acting upon the goals in the plans on the intention stack.

Definition 9.3.4 *(intentions)*
The set of *intentions* Int is defined by:

- If $\rho_1, \ldots, \rho_z \in \mathsf{PlanRule}$, then $[\rho_1 \ddagger \ldots \ddagger \rho_z] \in \mathsf{Int}$.

Note that in fact plan rules are put on a stack instead of plans. But as we will see below, only the plan - the body of the rule - plays a role of importance. Therefore, we will continue treating the plan rules in an intention simply as if they are plans.

The last concept incorporated in AgentSpeak(L) is a so-called *event.* Events have no 3APL analogue. An event is a pair that consists of a triggering event and an intention. The intention component of an event is used for administrative purposes and indicates which intention triggered the event. The triggering event component specifies the type of the event.

Definition 9.3.5 *(events)*
The set of *events* Event is defined by:

- If $e \in \mathsf{TrigEv}$ and $i \in \mathsf{Int}$, then $\langle e, i \rangle \in \mathsf{Event}$.

Events play a role in the construction of intentions. When - during the execution of an intention - the next thing that an agent has to accomplish is an achievement goal $!\phi$, a corresponding event is generated. The triggering event component of the event in that case is of the form $+!\phi$ and the intention i is the same as the intention that generated the event. The event indicates that a suitable plan to achieve the goal has to be found. A suitable (instance) of a plan rule is searched for, and in case such a plan (rule) is found, it is added to the intention stack i. If the plan has been executed completely, the plan (rule) is removed again from the intention.

All the ingredients of an AgentSpeak(L) agent have now been introduced. An AgentSpeak(L) agent is defined by its beliefs, intentions, and plan rules. During the execution of AgentSpeak(L) agents also a set of events that has been generated is maintained in the state of the agent. Initially, we require that the set of events is empty and that all intentions are of the form $[+!\mathsf{true} : \mathsf{true} \leftarrow h_1;\ \ldots;\ h_n]$. These requirements do not restrict the operation of agents and are quite natural initialisation conditions. The first condition states that no events have been generated yet when the execution of an agent is started. The second condition states that an agent may have adopted a number of plans to act upon initially, but no stack of more than one plan has been created yet.

Definition 9.3.6 *(AgentSpeak(L) agents)*
An *AgentSpeak(L) agent* is a tuple $\langle E, B, P, I\rangle$, where

- $E \subseteq \mathsf{Event}$ is a set of *events*,
- $B \subseteq \mathsf{Lit}$ is a set of ground *belief literals*,
- $P \subseteq \mathsf{PlanRule}$ is a set of *plan rules*, and
- $I \subseteq \mathsf{Int}$ is a set of *intentions*.

such that (i) $E = \varnothing$, and (ii) all $i \in I$ are of the form $[+!\mathsf{true} : \mathsf{true} \leftarrow h_1;\ \ldots;\ h_n]$.

The definition of AgentSpeak(L) agents that we have given deviates to some extent from that in (Rao 1996*a*). First, the effect of action execution in (Rao 1996*a*) is that any action which has been executed is stored in a set in the mental state of the agent. This set of actions that keeps record of the actions that the agent executes plays no other role in the operational semantics. The idea, supposedly, is that actions are executed by an 'external' system which is not specified in the agent semantics. The role played by actions in the operational semantics of AgentSpeak(L), however, is minimal. As an alternative, we can use the type of action semantics used for 3APL and define actions as updates on the belief base of an agent. If the agent needs to keep track of the actions that it has executed, in an update semantics it can store this information in its belief base.

Secondly, we have deviated from (Rao 1996*a*) since we have not included the three selection functions for selecting events, intentions and plans in the definition of an agent. These functions are used in (Rao 1996*a*) to define a (deterministic) interpreter for the execution of agent programs. They are used for control aspects which can be viewed as part of a meta layer that determines what an agent should do in case there are multiple options. Elsewhere (Hindriks et al. 1999*b*), we have argued that such features of agent control are better described at a different level. We think that it is better that such features are not included in the definition of the operational semantics of an agent language. Instead, they can be viewed as a feature of an interpreter that implements the agent language. The selection functions from (Rao 1996*a*) thus viewed define part of a control structure for an interpreter for AgentSpeak(L) (cf. also Hindriks et al. (1999*b*) and Rao (1996*a*)) that is not considered here.

9.3.2 Semantics of AgentSpeak(L)

The semantics of AgentSpeak(L) is defined by a somewhat different formalism than that of a transition system. In our presentation of the semantics of AgentSpeak(L), we have tried to stay as close as possible to the original presentation in (Rao 1996*a*). However, the definition of the semantics has been modified and extended at several places to correct for certain omissions. At the appropriate places we will comment on these changes.

The operational semantics of AgentSpeak(L) is defined by a so-called *proof system* in (Rao 1996*a*). Such a proof system is used to derive the possible computation steps of an agent. Proof systems are quite similar to transition systems. A proof system for AgentSpeak(L) consists of a set of proof rules that define a derivability relation $\vdash$. $\vdash$ is a relation on AgentSpeak(L) configurations. Such configurations are also called *BDI configurations.*

Definition 9.3.7 *(BDI configuration)*
A *BDI configuration* is a tuple $\langle E, B, I \rangle$, where

- $E \subseteq \mathsf{Event}$ is a set of events,
- $B \subseteq \mathsf{Lit}$ is a set of ground belief literals, and
- $I \subseteq \mathsf{Int}$ is a set of intentions.

A BDI configuration consists of a set of events, a set of beliefs, and a set of intentions. Notice that the belief base in a configuration is required to be closed. A BDI configuration consists of components of an agent that can change during the execution of that agent. Other components - like the plan rules associated with an agent - that remain fixed are not included in a configuration.

Each proof rule in a proof system is of the form

$$\frac{\langle E, B, I \rangle}{\langle E', B', I' \rangle} \quad \textit{conditions}$$

Both the premise and the conclusion of a rule are configurations. In contrast, premises and conclusions in a transition system are transitions. A proof rule corresponds to a type of computation step.

To provide the semantics of agent execution, in the remainder of this chapter, we will assume that a fixed set of plan rules P is associated with an agent. Furthermore, we assume that a function $\mathcal{T} : \mathsf{Act} \times \wp(\mathsf{Lit}) \to \wp(Lit)$ that specifies the update semantics of basic actions is given.

The first proof rule in the proof system for AgentSpeak(L) is a rule that defines how events should be handled. Recall that events are generated to deal with achievement goals, that is, in order to find a plan that supplies the means to achieve the goal. The triggering event component of the event denotes the achievement goal for which a plan must be found. In case an appropriate plan is found, the event is removed, the plan (rule) is pushed on top of the intention component of the event and this new intention is added to the intention base of the agent.

Definition 9.3.8 *(proof rule* IntendMeans*)*

$$\frac{\langle\{\ldots,\langle +!p(\vec{t}),[\rho_1\ddagger\ldots\ddagger\rho_z]\rangle,\ldots\},B,I\rangle}{\langle\{\ldots\},B,I\cup\{[\rho_1\ddagger\ldots\ddagger\rho_z\ddagger\rho]\eta\theta\}\rangle}$$

such that

- $\rho_z = e : \phi \leftarrow !p(\vec{t});\ g_2;\ \ldots;\ g_l$,
- $\rho = +!p(\vec{s}) : \psi \leftarrow h_1;\ \ldots;\ h_n$ is a variant of a plan rule in P, such that the variables that occur in ρ do not occur in either the event base or intention base,
- η is a most general unifier such that $p(\vec{t}) = p(\vec{s})\eta$, and
- $B \models \psi\eta\theta$, for a ground substitution θ such that $dom(\theta) \subseteq \mathsf{Free}(\varphi\theta)$.

The substitution η unifies the triggering event $+!p(\vec{s})$ in the head of the plan rule and the triggering event $+!p(\vec{t})$ in the event. Because η unifies both triggering events, the plan $h_1;\ \ldots;\ h_n$ in plan rule ρ may be a suitable plan for establishing achievement goal $!p(\vec{t})$. To check the suitability of the plan rule, an appropriate instantiation of the context ψ of the rule must be found. In case a substitution θ can be found such that $B \models \psi\eta\theta$, the plan may be selected. The function of the substitution θ is to retrieve specific parameters from the belief base to instantiate the plan with. A new intention is constructed by pushing the plan on the intention component of the event. The composed substitution $\eta\theta$ is used to instantiate variables in this new intention (and thus variables in the plan).

There is one important difference between the rule IntendMeans as it has been presented here and as it originally was presented in (Rao 1996*a*). The difference concerns the renaming of variables. In the proof rule, a *variant* of a plan rule ρ taken from P is used. In a variant all variables have been renamed (cf. 3.6.1). Such renaming is necessary to avoid interference between variables that are introduced by the plan rule and those which already occur in the original intention. The mechanism is completely analogous to the renaming mechanism of 3APL and is discussed in more detail in chapter 3. Another modification concerns the application of the substitution $\eta\theta$. This substitution should be applied to the complete intention and not only to the last plan rule that is pushed onto the intention. Otherwise, the mechanism for parameter passing would be limited for no good reason.

The proof rule IntendMeans specifies how so-called *internal* events should be handled. In (Rao 1996*a*), another rule is presented that deals with so-called *external* events. External events are events in which the intention component is absent. An intention called the *true intention* is constructed to model this situation. We have not included a rule for external events, however, since such events are not generated by the other rules in the proof system.

The generation of an (internal) event is formally specified by the proof rule below. It is the only proof rule in the proof system in which a new event is created and added to the set of events. An event is generated in case - during the execution of a plan on an intention j - an achievement goal must be executed. To achieve such a goal $!p(\vec{t})$ a plan has to be found, and an event with a corresponding triggering event $+!p(\vec{t})$ and corresponding intention j that gave rise to the event is generated.

Definition 9.3.9 *(proof rule* ExecAch*)*

$$\frac{\langle E, B, \{\ldots, j, \ldots\}\rangle}{\langle E \cup \{\langle +!p(\vec{t}), j\rangle\}, B, \{\ldots\}\rangle}$$

such that

- $j = [\rho_1 \ddagger \ldots \ddagger \rho_{z-1} \ddagger (e : \phi \leftarrow !p(\vec{t});\ h_2;\ \ldots;\ h_n)]$.

Note that the intention j is removed from the intention base, which is indicated by the fact that the intention base $\{\ldots, j, \ldots\}$ is updated to $\{\ldots\}$. This is different from (Rao 1996*a*) where - if an event is generated - the intention is not removed from the intention base. We do not quite understand why the intention is not removed in the presentation of the semantics in (Rao 1996*a*). If we trace the generation of an event and the consecutive handling of an event by the rule IntendMeans, we see that the original intention is modified by pushing a new plan on top of it. By also keeping a copy of the original intention in the intention base, the suggestion is raised that this intention still needs to be dealt with. For this reason, we have removed the intention from the intention base. In (d'Inverno & Luck 1998), a somewhat different approach to the same problem is taken. A status indicator is associated with an intention and in case the intention has generated an event, the intention is *suspended.*

Proof rule ExecAch specifies what happens when an achievement goal is executed. The next two proof rules specify what happens when an action or a test goal is executed. As we discussed previously, we associate an update semantics with actions here and do not simply add it to a set of actions that needs to be performed by some external system. The update semantics generalises the semantics of actions in (Rao 1996*a*).

In an update semantics for actions, the execution of an action results in updating the belief base in correspondence with the specific update semantics that is associated with the action. The type of update associated with an action is given by the transition function $\mathcal{T}$. After updating the beliefs of the agent, the action is subsequently removed from the intention.

Definition 9.3.10 *(proof rule* ExecAct*)*

$$\frac{\langle E, B, \{\ldots, [\rho_1 \ddagger \ldots \ddagger (e : \phi \leftarrow \mathsf{a}(\vec{t});\ h_2;\ \ldots;\ h_n)], \ldots\}\rangle}{\langle E, B', \{\ldots, [\rho_1 \ddagger \ldots \ddagger (e : \phi \leftarrow h_2;\ \ldots;\ h_n))], \ldots\}, \rangle}$$

such that

- $\mathcal{T}(\mathsf{a}(\vec{t}), B) = B'$.

A test goal $?p(\vec{t})$ is executed by checking whether an *instance* of the formula $p(\vec{t})$ is implied by the belief base of the agent. A test goal thus may retrieve data from the belief base. This data is recorded in a ground substitution θ. After the test has been successfully performed, the test goal is removed from the intention. The substitution θ is used to instantiate variables with the computed data in the remaining part of the intention.

Definition 9.3.11 *(proof rule* ExecTest*)*

$$\frac{\langle E, B, \{\ldots, [\rho_1 \ddagger \ldots \ddagger (e : \phi \leftarrow ?p(\vec{t});\ h_2;\ \ldots;\ h_n)], \ldots\}\rangle}{\langle E, B, \{\ldots, [\rho_1 \ddagger \ldots \ddagger (e : \phi \leftarrow h_2;\ \ldots;\ h_n)]\theta, \ldots\}\rangle}$$

such that

- $B \models p(\vec{t})\theta$, and
- γ is a ground substitution such that $\mathrm{dom}(\theta) = \mathsf{Free}(p(\vec{t}))$.

Again, there are some minor differences between (Rao 1996*a*) and our definition of the execution of a test goal. The most important one is that the substitution θ is applied to the complete remaining intention and not just to the top element of the intention stack as is done in (Rao 1996*a*). The argument here is similar to that made above concerning the application of a substitution in the rule IntendMeans. The substitution θ is also required to be a ground substitution.

The previous proof rules dealt with the execution of intentions and the generation of events. The last two proof rules deal with the removal of a plan or an intention that has been completely executed, respectively. The rule CleanStack-Entry that is defined below was omitted in (Rao 1996*a*), as is also noted in (d'Inverno et al. 1998). The proof rule implements the notion of an intention that is said to have been *executed* from definition 16 in (Rao 1996*a*). It is used to remove a plan that has been completely executed. The entry occupied by such a plan is popped from the intention so that execution can continue with the remainder of the intention (which triggered the completed plan). Besides removing the plan, the achievement goal which gave rise to the plan at the next entry in the intention must also be removed. The reason is that completed plan execution indicates that the goal has been achieved. An empty body in a plan rule in an intention is used here to indicate that the plan has been executed completely.

Definition 9.3.12 *(proof rule* CleanStackEntry*)*

$$\frac{\langle E, B, \{\ldots, [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger (+!p(\vec{t}) : \phi \leftarrow)], \ldots\}\rangle}{\langle E, B, \{\ldots, [\rho_1 \ddagger \ldots \ddagger \rho_z'], \ldots\}\rangle}$$

such that

- $\rho_z = e : \psi \leftarrow !p(\vec{t});\ h_2;\ \ldots;\ h_n$,
- $\rho'_z = e : \psi \leftarrow h_2;\ \ldots;\ h_n$.

In the case that a completely executed plan is the only item left on the intention stack, the intention itself should be removed from the intention base. For this purpose, the proof rule CleanIntSet is introduced. Again, no such rule was provided in (Rao 1996*a*). The rule is provided here for completeness, although the effect of removing a completely executed intention does not change the observable behaviour of an agent as we will see below.

Definition 9.3.13 *(clean rule* CleanIntSet*)*

$$\frac{\langle E, B, \{\ldots, [+!p(\vec{t}) : \phi \leftarrow], \ldots\}\rangle}{\langle E, B, \{\ldots\}\rangle}$$

Summarising, the set of proof rules presented in this section defines the operational semantics of AgentSpeak(L) agents. Although we have based our description of these proof rules on (Rao 1996*a*), a number of changes to simplify or improve the presentation have been made. The proof rule IntendEnd from (Rao 1996*a*) has not been included in our presentation, because it deals with so-called external events which are never generated by a single agent. External events probably make more sense in a multi-agent setting, but their use in a single agent environment is less clear.

One of the more important differences between (Rao 1996*a*) and our presentation of the proof system concern the renaming of variables and the application of substitutions. In private communication (Rao 1997), these modifications were acknowledged to be improvements of the presentation in (Rao 1996*a*). The update semantics associated with actions generalises the semantics of (Rao 1996*a*). Finally, we provided proof rules for cleaning intention stacks in case a plan has been completely executed.

Computations and Observables

For the purpose of simulating AgentSpeak(L) agents by 3APL agents we need to formally define two concepts. In the framework of the previous chapter, the basic concepts that must be formalised are that of a computation and that of an observable.

To formalise AgentSpeak(L) computations, we can use the derivation relation $\vdash$ on BDI configurations of the AgentSpeak(L) proof system from the previous section. A proof rule in the proof system allows the derivation of a new configuration from an old one. The derivation relation is written as $C \vdash C'$. A derivation specifies a set of possible computations of an agent. This definition is completely analogous to that of a 3APL computation in definition 8.1.1.

Definition 9.3.14 *(BDI derivation)*
A *BDI derivation* is a finite or infinite sequence of BDI configurations, i.e. $C_0, \ldots, C_n, \ldots$, where each C_{i+1} is derivable from C_i according to a proof rule, i.e. $C_i \vdash C_{i+1}$.

The choice of observables is suggested by taking a closer look at the way AgentSpeak(L) agents are executed. An AgentSpeak(L) agent is intention-driven. By this we mean that an agent continuously is occupied with executing one of its intentions. As a result of executing its intentions, an agent updates its beliefs. As a consequence, an agent can be viewed as computing with beliefs. From this point of view, a computation then results in a sequence of belief bases such that each belief base is identical to or an update of the previous belief base. This perspective guides us in the choice of observables that is associated with an agent system. It suggests that taking the belief base of the agent as the observables of an agent system is a good choice.

Definition 9.3.15 *(observables)*
Let $\mathcal{C}^L$ be the set of all BDI configurations, and let $\langle E, B, I\rangle \in \mathcal{C}^L$ be such a configuration. The function $\mathcal{O}^L : \mathcal{C}^L \to \wp(\mathsf{Lit})$ is defined by $\mathcal{O}^L(\langle E, B, I\rangle) = B$. $\mathcal{O}^L$ yields the *observable* of a configuration.

The most important parameter in the framework for bisimulation of the previous chapter, the observables, has been decided by this choice. Notice that we use only state-based observables here. For our purpose, this choice means that we must show that agents from two different agent languages are capable of producing the same sequences of (observable) belief bases. To show this, we need to prove that there exists a natural translation from AgentSpeak(L) agents to agents in another language, such that the translation of an AgentSpeak(L) agent to this other language simulates that agent. In that case, the latter language has at least the same expressive power as the former.

Notice that under this choice of observables the decoder function δ in our framework (cf. 8.3.1) trivialises. δ can be identified with the identity function since belief bases, which we have choosen as observable for AgentSpeak(L) agents, are also part of 3APL agents. By choosing the same observables for 3APL, we then do not need to 'decode' the 3APL observables anymore into AgentSpeak(L) observables.

9.3.3 The syntax of AgentSpeak(1)

To show that AgentSpeak(L) agents can be simulated by agents which do not use events in their operation, we now introduce a language that is very similar to AgentSpeak(L) but does not include events. The definition of the syntax of this language which we call AgentSpeak(1) is exactly the same as that for AgentSpeak(L) except that AgentSpeak(1) does not include events. AgentSpeak(1) thus is a proper subset of AgentSpeak(L).

The first step in showing that AgentSpeak(1) has at least the same expressive power as AgentSpeak(L) then is to construct a compositional translation function τ_1 that maps AgentSpeak(L) agents onto AgentSpeak(1) agents. A natural candidate for the function τ_1 is the function that maps all syntactic categories of AgentSpeak(L) to the same categories in AgentSpeak(1). This works for all AgentSpeak(L) expressions except, of course, for AgentSpeak(L) events. Events need to be mapped onto some other expression since events are not present in AgentSpeak(1).

The basic idea is to map events onto AgentSpeak(1) intentions. The idea to map events onto intentions is explained as follows. Events are used to indicate that a plan to achieve some achievement goal has to be found. The creation of an event thus forms an intermediate step in the process of creating a new intention. The proof rule of AgentSpeak(L) that is important here is the rule IntendMeans since it deals with events. An event is processed by pushing a suitable plan onto the intention component of the event. Events thus are used in an intermediate step in which a stack operation on an intention is performed. It now becomes clear how we can do without events. By modifying the rule ExecAch in such a way that it already incorporates the stack operation that pushes a new plan on an intention, events are no longer necessary.

Definition 9.3.16 *(translation function τ_1)*
The translation function τ_1 that translates AgentSpeak(L) expressions into AgentSpeak(1) expressions is defined as the identity, except for events, for which it is defined by:

- $\tau_1(\langle e, j\rangle) = j$.

It is easy to see that τ_1 is compositional. AgentSpeak(1) agents are similar to AgentSpeak(L) agents but do not have an event base.

Definition 9.3.17 *(AgentSpeak(1) agent)*
An *AgentSpeak(1) agent* is a tuple $\langle B, P, I\rangle$ where

- $B \subseteq \mathsf{Lit}$ is a set of ground belief literals,
- $P \subseteq \mathsf{PlanRule}$ is a set of plans, and
- $I \subseteq \mathsf{Int}$ is a set of intentions.

such that all $i \in I$ are of the form $[+!\mathsf{true} : \mathsf{true} \leftarrow h_1;\ \ldots;\ h_n]$.

The translation of an AgentSpeak(L) agent $\langle E, B, P, I\rangle$ to an AgentSpeak(1) agent by τ_1 is defined as follows: $\tau_1(\langle E, B, P, I\rangle) = \langle B, P, I \cup \tau_1(E)\rangle$. Here, we assume that τ_1 is lifted to sets of expressions point-wise, i.e. $\tau_1(S) = \{\tau_1(s) \mid s \in S\}$. It is easy to verify that this mapping conforms with the compositionality requirements that were outlined in the previous chapter in section 8.4.

9.3.4 Semantics of AgentSpeak(1)

The basic setup for the translation bisimulation has now been introduced. However, we still need to define the formal semantics for AgentSpeak(1). The operational semantics for AgentSpeak(1) can almost completely be copied from that of AgentSpeak(L). The main effort is to formally define the idea introduced above to simulate events. Apart from this modification of the semantics for AgentSpeak(L), we also replace the style of presentation of the semantics. Instead of the proof systems used to specify the semantics of AgentSpeak(L), we will use a transition system to specify the semantics of AgentSpeak(1). The transition system for AgentSpeak(1) defines a transition relation $\longrightarrow_1$. This relation is the analogue of AgentSpeak(1) for the derivation relation $\vdash$ of AgentSpeak(L). The so-called BDI configurations of AgentSpeak(L) are replaced with slightly different configurations for AgentSpeak(1). An AgentSpeak(1) configuration is a pair that consists of beliefs and intentions.

Definition 9.3.18 *(AgentSpeak(1) configuration)*
An *AgentSpeak(1) configuration* is a tuple $\langle B, I\rangle$, where

- $B \subseteq \mathsf{Lit}$ is a ground belief base,
- $I \subseteq \mathsf{Int}$ is a set of intentions.

By definition, an AgentSpeak(L) configuration is mapped onto an AgentSpeak(1) configuration by τ_1 as follows: $\tau_1(\langle E, B, I\rangle) = \langle B, I \cup \tau_1(E)\rangle$.

The main difference between the semantics for AgentSpeak(L) and that for AgentSpeak(1) concerns the handling of achievement goals. Whereas the rule $\mathsf{IntendMeans}$ for AgentSpeak(L) generates an event to deal with such a goal, in AgentSpeak(1) such an event-producing step is absent. Instead, the AgentSpeak(1) transition rule for achievement goals immediately pushes an appropriate plan for the goal on the intention of which it is a part. In this way, the need for events is circumvented. In a sense, the transition rule for achievement goals combines the rules $\mathsf{IntendMeans}$ and $\mathsf{ExecAch}$ into a single rule.

Definition 9.3.19 *(transition rule for achievement goals)*
Let η be a most general unifier for $!p(\vec{t})$ and $!p(\vec{s})$ such that $p(\vec{t}) = p(\vec{s})\eta$, and θ be a ground substitution.

$$\frac{B \models \phi\eta\theta}{\langle B, \{\ldots, [\rho_1 \ddagger \ldots \ddagger \rho_z], \ldots\}\rangle \longrightarrow_1 \langle B, \{\ldots, [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\eta\theta, \ldots\}\rangle}$$

such that

- $\rho = +!p(\vec{s}) : \phi \leftarrow h_1;\ \ldots;\ h_n$ is a variant of a rule in P such that variables that occur in ρ do not occur in the intention base of the agent,
- $\rho_z = e : \phi \leftarrow !p(\vec{t});\ g_2;\ \ldots;\ g_l$.

The meaning of an intention is slightly changed by this transition rule. This illustrates our remark at the very end of the previous chapter that small changes in the meaning of one and the same construct in the source and target language is allowed. As we show below, this shift in meaning still allows us to simulate AgentSpeak(L) agents with the benefit of reducing the complexity of the operation of AgentSpeak(L) since events are no longer needed.

The other four rules of AgentSpeak(1) are the transition rule variants of the rules $\mathsf{ExecAct}$, $\mathsf{ExecTest}$, $\mathsf{CleanStackEntry}$, and $\mathsf{CleanIntSet}$, respectively. In these rules, the set of events is dropped from the configurations.

Definition 9.3.20 *(transition rule for actions)*

$$\frac{\mathcal{T}(\mathsf{a}(\vec{t}), B) = B'}{\begin{array}{c}\langle B, \{..., [\rho_1 \ddagger ... \ddagger (e : \phi \leftarrow \mathsf{a}(\vec{t});\ h_2;\ ...;\ h_n)], ...\}\rangle \longrightarrow_1 \\ \langle B', \{..., [\rho_1 \ddagger ... \ddagger (e : \phi \leftarrow h_2;\ ...;\ h_n))], ...\}\rangle\end{array}}$$

Definition 9.3.21 *(transition rule for test goals)*
Let θ be a ground substitution such that $\mathrm{dom}(\theta) = \mathsf{Free}(p(\vec{t}))$.

$$\frac{B \models p(\vec{t})\theta}{\begin{array}{c}\langle B, \{..., [\rho_1 \ddagger ... \ddagger (e : \phi \leftarrow ?p(\vec{t});\ h_2;\ ...;\ h_n)], ...\}\rangle \longrightarrow_1 \\ \langle B, \{..., [\rho_1 \ddagger ... \ddagger (e : \phi \leftarrow h_2;\ ...;\ h_n)]\theta, ...\}\rangle\end{array}}$$

Definition 9.3.22 *(transition rule to clean a stack entry)*

$$\frac{}{\langle B, \{\ldots, [\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger (+!p(\vec{t}) : \phi \leftarrow)], \ldots\}\rangle \longrightarrow_1 \langle B, \{\ldots, [\rho_1 \ddagger \ldots \ddagger \rho'_z], \ldots\}\rangle}$$

such that

- $\rho_z = e : \psi \leftarrow !p(\vec{t});\ h_2;\ \ldots;\ h_n,$
- $\rho'_z = e : \psi \leftarrow h_2;\ \ldots;\ h_n.$

Definition 9.3.23 *(transition rule to clean the intention base)*

$$\frac{}{\langle B, \{\ldots, [+!p(\vec{t}) : \phi \leftarrow], \ldots\}\rangle \longrightarrow_1 \langle B, \{\ldots\}\rangle}$$

Computations and Observables

The transition relation $\longrightarrow_1$ is the counterpart of AgentSpeak(1) for the derivation relation $\vdash$ of AgentSpeak(L). The choice of observables for AgentSpeak(1) is the same as that for AgentSpeak(L): the belief base of an AgentSpeak(1) configuration. As noted above, because of this choice we can use the identity function as decoder δ to map observables from AgentSpeak(1) to AgentSpeak(L) in the framework for translation bisimulation. It is an immediate consequence that δ is compositional.

Definition 9.3.24 *(observables)*
Let $\mathcal{C}^1$ be the set of all AgentSpeak(1) configurations. The function $\mathcal{O}^1 : \mathcal{C}^1 \to \wp(\mathsf{Lit})$ is defined by $\mathcal{O}^1(\langle B, I\rangle) = B$ for all $\langle B, I\rangle \in \mathcal{C}^1$. $\mathcal{O}^1$ yields the *observable* of an AgentSpeak(1) configuration.

9.3.5 AgentSpeak(1) bisimulates AgentSpeak(L)

To show that τ_1 is a translation bisimulation and that AgentSpeak(1) simulates AgentSpeak(L), we have to show that every computation step of an AgentSpeak(L) agent can be simulated by the translated AgentSpeak(1) agent, and vice versa. To prove this, there is only one more thing we have to decide: which steps may be considered as silent steps. In this chapter, all computation steps in which the observables are not changed are considered as silent steps. In the previous chapter, we argued that this is a very natural choice. That is, the silent steps of AgentSpeak(L) are those steps $C \vdash C'$ such that $\mathcal{O}^L(C) = \mathcal{O}^L(C')$ and the silent steps of AgentSpeak(1) are those steps $C \longrightarrow_1 C'$ such that $\mathcal{O}^1(C) = \mathcal{O}^1(C')$.

The parameters of the simulation problem are now completely fixed, and we need to prove that steps of AgentSpeak(L) agents can be simulated by AgentSpeak(1) agents, and vice versa. Since proof rules and transition rules define what steps an agent can perform, we need to check each of these rules in turn and see whether or not the step defined by a rule can be simulated. Since all AgentSpeak(1) rules except for the transition rule for achievement goals are nothing but notational variants of proof rules for AgentSpeak(L), the proof in these cases is rather straightforward.

It thus suffices to show that the computation steps defined by the proof rules IntendMeans and ExecAch can be simulated by AgentSpeak(1) steps, and that the transition rule for achievement goals can be simulated by AgentSpeak(L) steps. Each of these cases is dealt with in turn below.

Theorem 9.3.25
The computation step defined by the proof rule ExecAch of AgentSpeak(L) can be simulated by AgentSpeak(1) steps.

Proof: By inspection of the proof rule ExecAch and the definition of τ_1, it is easy to see that the AgentSpeak(L) configuration in the premise of the rule and the configuration of the conclusion of the rule are mapped onto the same AgentSpeak(1) configuration. The step defined by the proof rule ExecAch thus can be considered as a silent step and no AgentSpeak(1) step is needed to simulate this computation step. □

Theorem 9.3.26
The computation step defined by the proof rule IntendMeans of AgentSpeak(L) can be simulated by AgentSpeak(1) steps.

Proof: Let

$$A = \langle E \cup \{\langle +!p(\vec{t}), [\rho_1 \ddagger \ldots \ddagger \rho_z] \rangle\}, B, I \rangle, \text{ and}$$
$$A' = \langle E, B, I \cup \{[\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\eta\theta\} \rangle$$

such that $A \vdash A'$. Note that since IntendMeans is the only proof rule in which an event is removed from the event base, the step $A \vdash A'$ must have been a step defined by this rule.

The translation function τ_1 maps A onto $M = \tau(A) = \langle B, I \cup \tau_1(E) \cup \{[\rho_1 \ddagger \ldots \ddagger \rho_z]\}\rangle$. Since rule IntendMeans must have been used here, ρ_z must be of the form $e : \phi \leftarrow !p(\vec{t});\ g_2;\ \ldots;\ g_l$. Moreover, a plan rule ρ must provide a suitable plan for $!p(\vec{t})$ and substitutions η and θ must be available such that the head of ρ unifies with $!p(\vec{t})$ and the context of the rule is satisfied. In that case, we can apply the transition rule for achievement goals of AgentSpeak(1) to M and we obtain: $M' = \langle B, I \cup \tau_1(E) \cup \{[\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\eta\theta\}\rangle$. It is easy to see that $\tau_1(A') = M'$, since we have that $\tau_1(i\eta\theta) = \tau_1(i)\eta\theta$ for an intention i. $\Box$

Note that although the proof rule IntendMeans defines a silent step, the step that it defines cannot be simulated by performing no AgentSpeak(1) step at all. The reason is that $\tau_1(C) \neq \tau_1(C')$ if $C \vdash C'$ is a computation step derived by the proof rule IntendMeans.

Theorem 9.3.27
The computation step defined by the transition rule for achievement goals of AgentSpeak(1) can be simulated by AgentSpeak(L) steps.

Proof: To simulate steps derived by the transition rule for achievement goals by AgentSpeak(L), we may need to perform more than one AgentSpeak(L) step. This is due to the fact that the transition rule combines the proof rule IntendMeans as well as the proof rule ExecAch. Because steps defined by either of these proof rules are silent steps, it is allowed to use both types of computation steps.

Now, suppose that A is a BDI configuration, $M = \tau_1(A)$, and $M \longrightarrow_1 M'$ is a computation step derived by the transition rule for achievement goals. M and M' are both AgentSpeak(1) configurations. We need to show that there is a BDI configuration A' such that $A \vdash A'$ and $M' = \tau_1(A')$.

Since the rule for achievement goals has been used to derive $M \longrightarrow_1 M'$, the configurations M and M' must be of the form:

$$M = \tau(A) = \langle B, I \cup \{[\rho_1 \ddagger \ldots \ddagger \rho_z]\}\rangle, \text{ and },$$
$$M' = \langle B, I \cup \{[\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\eta\theta\}\rangle$$

By inspection of the translation function τ_1, we need to distinguish two cases. In the first case (i), the intention $[\rho_1 \ddagger \ldots \ddagger \rho_z]$ is a translation of an event in configuration A; in the second case (ii), it is a translation of an intention in configuration A. In both cases, ρ_z must be of the form $e : \phi \leftarrow !p(\vec{t});\ g_2;\ \ldots;\ g_l$. If *(ii)* is the case, then we have that proof rule ExecAch is applicable, and by performing the associated step we obtain a configuration of the first type (i). Recall that ExecAch defines a silent step and we are allowed to use it. We thus may assume that we have a configuration A of the form $\langle\{E \cup \{\langle +!p(\vec{t}), [\rho_1 \ddagger \ldots \ddagger \rho_z]\rangle\}, B, I\rangle$. Since the transition rule for achievement goals is applicable to $\tau_1(A)$, we know that all the conditions of the proof rule IntendMeans are also satisfied and we know that this rule is applicable. (Just compare the conditions of both rules.) By applying the rule IntendMeans, we obtain a new configuration $A' = \langle E, B, I \cup \{[\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\eta\theta\}\rangle$. It is easy to see that $\tau_1(A') = M'$, and we are done. □

We summarise the results of this section. Our aim was to prove that AgentSpeak(1) agents bisimulate AgentSpeak(L) agents and that (internal) events are eliminable from AgentSpeak(L). We thus have to verify that the conditions of eliminability defined in 8.4.4 are met. Because AgentSpeak(1) syntactically is a proper subset of AgentSpeak(L), condition **F3** of definition 8.4.4 is met. Moreover, since both the translation function τ_1 and the decoder δ are compositional, condition **F1** is met. Finally, it follows from the theorems above that τ_1 is a translation bisimulation. This corresponds to condition **F2** in definition 8.4.4. Taken together, this concludes the proof that events are eliminable from AgentSpeak(L), and we also have that AgentSpeak(1) has at least the same expressive power as AgentSpeak(L).

The proof that the translation function τ_1 is a translation bisimulation is a proof that AgentSpeak(1) has the expressive power of AgentSpeak(L), but not necessarily the other way around. However, we claim that AgentSpeak(L) also has the expressive power of AgentSpeak(1). Although we did not give a formal proof of this claim, the proof is easily derived from the proofs in this section. The translation function to show that AgentSpeak(L) has the same expressive power as AgentSpeak(1) - as can be easily verified - is the identity function. As a result, AgentSpeak(L) and AgentSpeak(1) have exactly the same expressive power.

9.4 The Transformation of Intentions to Goals

The second step in the proof that AgentSpeak(L) agents can be simulated by 3APL agents consists of showing that intentions can be simulated by 3APL goals. The proof continues with the language AgentSpeak(1). Since we already know that AgentSpeak(1) has the expressive power of AgentSpeak(L), it suffices to show that 3APL has the expressive power of AgentSpeak(1). Apart from some minor notational differences, it is easy to see that if the intentions of AgentSpeak(1) are replaced by 3APL goals, a new language that is a subset of 3APL is obtained. Note that triggering events in fact were already redundant in AgentSpeak(1), and can be replaced with the achievement goals they represent. The language obtained in this way is called AgentSpeak(2).

First, we introduce the syntax of AgentSpeak(2) and we define a compositional translation function τ_2 from AgentSpeak(1) to AgentSpeak(2). Then the semantics of AgentSpeak(2) is defined and we prove that τ_2 is a translation bisimulation.

9.4.1 Syntax of AgentSpeak(2)

The main difference between the syntax of AgentSpeak(1) and AgentSpeak(2) is that the latter does not have intentions. Also, a number of other changes in the syntax of AgentSpeak(1) goals and plan rules have been made to compensate for minor differences between the syntax of AgentSpeak(L) and 3APL. The symbol ! marking that a goal is an achievement goal in $!\phi$ is simply dropped. $?\phi$ is written as $\phi?$. And finally, a plan rule $+!p(\vec{t}) : \phi \leftarrow h_1;\ \ldots;\ h_n$ is written as $p(\vec{t}) \leftarrow \phi \mid h_1;\ \ldots;\ h_n$. Notice that due to this change in the syntax of plan rules, there is no need anymore for triggering events. Triggering events thus are also eliminated from AgentSpeak(L).

The syntax of AgentSpeak(2) beliefs and actions is identical to that of AgentSpeak(L) and is not repeated. A new type of goal is introduced in the language AgentSpeak(2): a composed goal that consists of a sequence of simple goals and actions. It is easy to see that the language AgentSpeak(2) (syntactically) is a subset of 3APL.

Definition 9.4.1 *(AgentSpeak(2) goals and plan rules)*
The syntax of AgentSpeak(2) goals and plan rules is defined as follows:

- If $\phi \in \mathsf{At}$, then $\phi \in \mathsf{Agoal}(2)$,
- If $\phi \in \mathsf{At}$, then $\phi? \in \mathsf{Agoal}(2)$,
- If $h_1, \ldots, h_n \in (\mathsf{Agoal}(2) \cup \mathsf{Act})$, then $h_1;\ \ldots;\ h_n \in \mathsf{Agoal}(2)$,
- If $\phi \in \mathsf{At}$, $b_1, \ldots, b_n$ are belief literals, and $h_1, \ldots, h_n \in (\mathsf{Agoal}(2) \cup \mathsf{Act})$, then $\phi \leftarrow b_1 \wedge \ldots \wedge b_n \mid h_1;\ \ldots;\ h_n \in \mathsf{PlanRule}(2)$.

The next step is to construct a compositional translation function τ_2 from AgentSpeak(1) agents to AgentSpeak(2) agents. The translation again is rather straightforward and mainly deals with small changes in syntax between AgentSpeak(1) and AgentSpeak(2). The main effort that has to be made is in the translation of intentions to AgentSpeak(2) goals. This translation requires that a stack of plans is mapped onto a sequence of actions and simple goals. The basic idea here is to pop each plan from the intention and put them in sequence. However, any achievement goals that triggered the addition of a plan on the stack should be removed. These goals are 'implemented' by these plans and are no longer required.

Technically, we define an auxiliary function τ_2', and use a function *body* to obtain the body of a plan rule. τ_2' removes the head h_1 of the sequence from any plan $h_1;\ \ldots;\ h_n$ in an intention. In the intentions that are constructed during execution, h_1 is an achievement goal. Since such goals are implemented by plans at the next higher entry in the stack except for the top element, they can be removed. The tails of the plans are put into sequence. Note that the top element of a stack is executed first and needs to be put at the front of the AgentSpeak(2) goal.

Example 9.4.2 As a very simple example of such a translation, consider the intention $[+!p : \phi \leftarrow !q;\ \mathsf{a}\ddagger+!q : \psi \leftarrow !r;\ \mathsf{b}]$. The last plan on this intention (the top element) must be executed first. This means that the achievement goal $!r$ must be executed (and should not be removed!) followed by b; after the plan has been executed it is removed and execution continues with the bottom element on the intention stack. Since the achievement goal $!q$ is implemented by the top plan, this achievement goal has to be removed in contrast with the achievement goal $!r$. What remains to be done is action a and, in the translation, we thus obtain a new goal $r;\ \mathsf{b};\ \mathsf{a}$ since the symbol ! is removed in the translation of an achievement goal.

We use E to denote the empty goal below, and stipulate that $h_1;\ \ldots;\ h_n;\ E$ is identified with $h_1;\ \ldots;\ h_n$

Definition 9.4.3 *(translation function τ_2)*
The translation function τ_2 that translates AgentSpeak(1) expressions into AgentSpeak(2) expressions is defined as the identity, except for the following cases:

- $\tau_2(!\phi) = \phi$,
- $\tau_2(?\phi) = \phi?$,
- $\tau_2(+!p(\vec{t}) : \phi \leftarrow h_1;\ \ldots;\ h_n) = p(\vec{t}) \leftarrow \phi \mid h_1;\ \ldots;\ h_n$,
- $\tau_2'([\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger (+!p(\vec{t}) : \phi \leftarrow h_1;\ h_2;\ \ldots;\ h_n)]) =$
  $h_2;\ \ldots;\ h_n;\ \tau_2'([\rho_1 \ddagger \ldots \ddagger \rho_z])$,
- $\tau_2'([]) = E$,
- $\tau_2([\rho_1 \ddagger \ldots \ddagger \rho_z]) = \mathit{body}(\tau_2(\rho_z));\ \tau_2'([\rho_1 \ddagger \ldots \ddagger \rho_{z-1}])$.

Again, it is easy to see that τ_2 is compositional. Notice that the composition $\tau_1 \circ \tau_2$ translates AgentSpeak(L) expressions to AgentSpeak(2) expressions.

AgentSpeak(2) agents consist of a belief base, a goal base and a plan base. The goal base of an AgentSpeak(2) agent replaces the intention base of an AgentSpeak(1) agent.

Definition 9.4.4 *(AgentSpeak(2) agent)*
An *AgentSpeak(2) agent* is a tuple $\langle B, P, G\rangle$ where

- $B \subseteq \mathsf{Lit}$ is a set of ground belief literals,
- $P \subseteq \mathsf{PlanRule}(2)$ is a set of plan rules, and
- $G \subseteq \mathsf{Agoal}(2)$ is a set of goals.

The translation function τ_2 is extended to agents and an AgentSpeak(1) agent $\langle B, P, I\rangle$ is mapped onto the AgentSpeak(2) agent $\langle B, \tau_2(P), \tau_2(I)\rangle$. τ_2 is lifted point-wise to sets.

9.4.2 Semantics of AgentSpeak(2)

The semantics of AgentSpeak(2) is that of 3APL, but it can also be looked upon as a transformation of the AgentSpeak(1) semantics. The transformation of the AgentSpeak(1) semantics to the AgentSpeak(2) semantics then involves the replacement of rules that deal with intentions to rules that deal with composed goals. The intention base of AgentSpeak(1) configurations is replaced with a goal base in AgentSpeak(2) configurations.

Definition 9.4.5 *(AgentSpeak(2) configuration)*
An *AgentSpeak(2) configuration* is a pair $\langle B, G\rangle$, where

- $B \subseteq \mathsf{Lit}$ is a closed belief base, and
- $G \subseteq \mathsf{Agoal}(2)$ is a set of goals.

By definition, an AgentSpeak(1) configuration is mapped onto an AgentSpeak(2) configuration by τ_2 as follows: $\tau_2(\langle B, I\rangle) = \langle B, \tau_2(I)\rangle$.

The main difference between the transition rules of AgentSpeak(1) and those of AgentSpeak(2) is that the transition rules of AgentSpeak(2) exploit the recursive capabilities of transition systems. That is, the transition rules of AgentSpeak(2) are defined on the syntactic structure of agents and goals. The rules decompose a composed goal to its elementary parts, and the semantics of a composed goal is derived from the rules for the elementary parts. For example, the semantics of a sequential goal is derived from the semantics of the head of the sequential goal.

The transition rules for AgentSpeak(2) are (almost completely) the same as those for 3APL (cf. chapter 3), but for completeness we will present them here once again. The transition rule for agent execution selects a goal that can be executed or that can be modified by applying a plan rule. This transition rule defines a transition relation $\longrightarrow_2$ on AgentSpeak(2) configurations $\langle B, G\rangle$. The transition relation $\longrightarrow_2$ is derived from another transition relation denoted by $\longrightarrow$. $\longrightarrow$ is a relation on pairs $\langle B, \pi\rangle$ of a belief base and a single goal.

Definition 9.4.6 *(transition rule for agent execution)*

$$\frac{\langle B, \pi\rangle_V \longrightarrow_\theta \langle B', \pi'\rangle}{\langle B, \{\ldots, \pi, \ldots\}\rangle \longrightarrow_2 \langle B', \{\ldots, \pi', \ldots\}\rangle}$$

where V is the set of variables in $\{\ldots, \pi, \ldots\}$.

To handle an achievement goal, a plan must be found for achieving the goal. In case such a plan can be found, the achievement goal is replaced by this plan. The transition rule for achievement goals is the AgentSpeak(2) analogue for the rule $\mathsf{IntendMeans}$ of AgentSpeak(L).

Definition 9.4.7 *(transition rule for achievement goals)*
Let η be a most general unifier such that $p(\vec{t}) = p(\vec{s})\eta$, and θ be a ground substitution such that $dom(\theta) \subset \mathsf{Free}(\phi\eta)$.

$$\frac{B \models \phi\eta\theta}{\langle B, p(\vec{t})\rangle_V \longrightarrow_\theta \langle B, (h_1;\ \ldots;\ h_n)\eta\theta\rangle}$$

such that

- $p(\vec{s}) \leftarrow \phi \mid h_1;\ \ldots;\ h_n$ is a variant of a rule in P such that variables in this rule do not occur in V.

The execution of a basic action consists in an update on the belief base. $\varnothing$ denotes the empty substitution and the symbol E denotes the successful termination of a goal.

Definition 9.4.8 *(transition rule for actions)*

$$\frac{\mathcal{T}(\mathsf{a}(\vec{t}), B) = B'}{\langle B, \mathsf{a}(\vec{t})\rangle_V \longrightarrow_\varnothing \langle B', E\rangle}$$

As before, a test is used to inspect the belief base.

Definition 9.4.9 *(transition rule for tests)*
Let θ be a ground substitution such that $\mathrm{dom}(\theta) = \mathsf{Free}(p(\vec{t}))$.

$$\frac{B \models p(\vec{t})\theta}{\langle B, p(\vec{t})?\rangle_V \longrightarrow_\theta \langle B, E\rangle}$$

The rule for a sequence of actions and simple goals selects the head of the sequence for execution, executes it and updates both the belief base and the goal accordingly.

Definition 9.4.10 *(transition rule for a sequential goal)*
Let θ be a substitution.

$$\frac{\langle B, h_1\rangle \longrightarrow_\theta \langle B', h_1'\rangle}{\langle B, h_1;\ h_2;\ \ldots;\ h_n\rangle_V \longrightarrow_\theta \langle B', (h_1';\ h_2;\ \ldots;\ h_n)\theta\rangle}$$

9.4.3 Computations and Observables

An AgentSpeak(2) computation is a finite or infinite sequence of AgentSpeak(2) configurations where each consecutive pair is related by the transition relation $\longrightarrow_2$. As before in the case of AgentSpeak(1), the choice of observables is the same as that for AgentSpeak(L): the belief base of an AgentSpeak(2) configuration. Again, the decoder δ is the identity function.

Definition 9.4.11 *(observables)*
Let $\mathcal{C}^2$ be the set of all AgentSpeak(2) configurations. The function $\mathcal{O}^2 : \mathcal{C}^2 \to \wp(\mathsf{Lit})$ is defined by $\mathcal{O}^2(\langle B, G\rangle) = B$. $\mathcal{O}^2$ yields the *observable* of an AgentSpeak(2) configuration.

9.4.4 AgentSpeak(2) simulates AgentSpeak(1)

To show that τ_2 is a translation bisimulation and that AgentSpeak(2) simulates AgentSpeak(1), we have to show that every computation step of an AgentSpeak(1) agent can be simulated by the translated AgentSpeak(2) agent, and vice versa. As before, we will use the criterion that transitions in which observables are not changed are considered to be silent steps. The proof consists of checking whether computation steps defined by the different transition rules can be simulated. The proof for most of the transition rules is rather straightforward. Therefore, we will only provide a proof for the most complex case, namely the case that a plan rule is applied in a computation step.

Notice that the computation steps for cleaning intention stacks and the intention base are silent steps which require no AgentSpeak(2) step at all to simulate these steps. The reason is that the translation function τ_2 maps the configuration in the premise of these rules onto the same AgentSpeak(2) configuration as the conclusion of these rules.

Theorem 9.4.12
The computation step defined by the transition rule for achievement goals of AgentSpeak(1) can be simulated by an AgentSpeak(2) step.

Proof: Suppose that the AgentSpeak(1) step $A \longrightarrow_1 A'$ is derived by the transition rule for achievement goals of AgentSpeak(1). In that case, A is of the form $\langle B, I \cup \{[\rho_1 \ddagger \ldots \ddagger \rho_z]\}\rangle$ and A' must be of the form $\langle B, I \cup \{[\rho_1 \ddagger \ldots \ddagger \rho_z \ddagger \rho]\eta\theta\}\rangle$. Moreover, ρ_z and ρ must be of the form $e : \phi \leftarrow !p(\vec{t});\ g_2;\ \ldots;\ g_l$, and $+!p(\vec{s}) : \psi \leftarrow h_1;\ \ldots;\ h_n$, respectively.

The translations of both agent A and A' are:

$$\begin{aligned}
&M = \tau_2(A) = \langle B, \tau_2(I) \cup \{p(\vec{t});\ \tau_2(g_2);\ \ldots;\ \tau_2(g_l);\ \tau_2'([\rho_1 \ddagger \ldots \rho_{z-1})\}\rangle\\
&M' = \tau_2(A') =\\
&\qquad \langle B, \tau_2(I) \cup \{(h_1;\ \ldots;\ h_n;\ \tau_2(g_2);\ \ldots;\ \tau_2(g_l);\ \tau_2'([\rho_1 \ddagger \ldots \ddagger \rho_{z-1}]))\eta\theta\}\rangle
\end{aligned}$$

It is easy to check that by means of the transition rule for agent execution, the transition rule for a sequential goal, and the transition rule for achievement goals of AgentSpeak(2) we can derive that $M \longrightarrow_2 M'$. Use the fact that the conditions required in the transition rule for achievement goals are the same in AgentSpeak(1) and AgentSpeak(2). □

Theorem 9.4.13 The computation step defined by the transition rule for achievement goals of AgentSpeak(2) can be simulated by an AgentSpeak(1) step.

Proof: Let $M = \tau_2(A) = \langle B, G\rangle$, and suppose that $M \longrightarrow_2 M'$ is derived by the transition rule for achievement goals of AgentSpeak(2). This implies that a goal of the form $p(\vec{t});\ \pi$ must be an element of the goal base G for some (possibly empty) sequence π. By inspection of the translation function τ_2, this goal must have originated from an intention in the configuration A. But in that case, we can apply the transition rule for achievement goals of

AgentSpeak(1) to transform the intention and add the same plan that was used in the AgentSpeak(2) computation step $M \longrightarrow_2 M'$ to the stack of plans. This step results in a new configuration A'. It is not difficult to check that the translation function τ_2 maps this configuration to M', that is, $\tau_2(A') = M'$, and we are done. □

We summarise the results of this section. Our aim was to show that AgentSpeak(2) agents can simulate AgentSpeak(1) agents. We need to check the requirements of definition 8.4.1 in which the notion of relative expressive power is formally defined. First, since both the translation function τ_2 as well as the decoder δ are compositional, condition **E1** is met. The second condition **E2** is met by theorems 9.4.12 and 9.4.13 which show that τ_2 is a translation bisimulation. Taken together, this concludes the proof that AgentSpeak(2) has at least the same expressive power as AgentSpeak(1).

As before, we believe that AgentSpeak(1) also has the expressive power of AgentSpeak(2). In that case, both languages have the same expressive power. Since AgentSpeak(2) is a proper subset of 3APL, by transitivity of the expressiveness relation we obtain that 3APL has at least the same expressive power as AgentSpeak(2).

Since AgentSpeak(2) is a proper subset of 3APL, 3APL may still have more expressive power than AgentSpeak(L). In particular, there is one feature of 3APL that has not been used in the simulation of AgentSpeak(L) agents, namely practical reasoning rules. Practical reasoning rules allow for a more general type of goal revision (cf. also chapter 4). It is not easy to see how this feature can be simulated by AgentSpeak(L), and we conjecture that 3APL indeed has more expressive power.

9.5 Conclusion

A formal comparison of AgentSpeak(L) with 3APL showed that 3APL has at least the same expressive power of AgentSpeak(L) and probably has more expressive power. The proof of this claim focused on the main conceptual differences between both languages: the concept of an event and of an intention which are part of AgentSpeak(L) but not of 3APL. It was shown that the former can be eliminated from AgentSpeak(L) and the latter can be translated into 3APL goals.

A number of 3APL features were not needed to simulate AgentSpeak(L) agents. In particular, a number of (imperative) programming constructs like parallel composition and non-deterministic choice have not been used. Another feature that has not been used are the more complex practical reasoning rules of 3APL. In the bisimulation proofs, we only used plan rules. Practical reasoning rules may have a more complex structure than the plan rules of AgentSpeak(L). A practical reasoning rule also provides the means for modifying composed plans or goals of an agent (cf. chapter 4), and do not just provide the plans for achievement goals.

The lack of a number of regular programming constructs should not be viewed as too great a difference, since it is easy enough to extend AgentSpeak(L) with such constructs. As we noted above already, however, the practical reasoning rules of 3APL that provide for a more general revision mechanism of composed goals (to which intentions are translated) is of more interest. We believe it is not possible to define a translation bisimulation that translates arbitrary 3APL agents including this feature into AgentSpeak(L) agents. Therefore, we conjecture that 3APL has strictly more expressive power than AgentSpeak(L).

From the bisimulation result, we can conclude that the notions of events and intentions play a similar role and in fact can be identified. This follows from the the fact that both events and intentions are simulated by 3APL goals. Events as well as intentions are mapped onto goals by the translation function $\tau_1 \circ \tau_2$.

As we showed, events can be eliminated from AgentSpeak(L) without reducing its expressive power. Intentions cannot be removed without reducing the expressive power of the language. However, the simulation results do show that there is no need to maintain a complete stack of plan rules as is done in AgentSpeak(L). Intentions consists of a lot of redundant information that is not used in the operation of an agent. That this extra information can be removed is shown by the fact that intentions are simulated by 3APL goals which do not contain this information. The bookkeeping for which both events and intentions are used, therefore, only complicates the semantics of AgentSpeak(L). Since there is no loss of expressiveness, stacks and (triggering) events can be viewed as one possible implementation of the agent language AgentSpeak(L), but preferably should not be incorporated into the semantics of the language.

Of course, one could argue that it is useful for an agent to keep track of the achievement goals that it is pursuing and the plans it is trying to use to achieve these goals as is done in an intention. For example, if a plan fails to achieve a goal, the plan could be dropped, the old goal could be retrieved (from the next entry in the intention structure) and a new plan could be searched for. Although this argument is valid, the complexity both from a theoretical and a practical perspective of this kind of 'backtracking' is not to be underestimated and raises questions concerning the viability of the idea. First of all, new types of rules would have to be introduced that can be used to change intention structures in a way imagined here. These new rules would make the semantics considerably more complex. Another problem is that it is not (yet) clear when and how to use this type of 'backtracking'. Moreover, there is a more simple alternative as we showed in chapter 4. Practical reasoning rules of 3APL can be used in a very flexible way to achieve similar goals. These rules are a suitable and practical means to incorporate a goal revision mechanism into an agent language.

Finally, apart from the operational semantics for AgentSpeak(L), in (Rao 1996*a*) also an algorithm for an interpreter for AgentSpeak(L) is defined. This algorithm specifies in which order the proof rules should be used to execute AgentSpeak(L) agents. For example, in every cycle of the interpreter first an event is processed and then an intention is processed. We have looked in more detail at the specification of an interpreter for AgentSpeak(L) in (Hindriks et al. 1999*b*). In this paper, the results obtained in this chapter that events and intentions can be translated into goals have been used.

CHAPTER 10

An Embedding of ConGolog in 3APL

An interesting alternative for agent programming, based on a logical perspective, is offered by the concurrent language ConGolog (Giacomo et al. 2000). In this chapter, we present a formal comparison of ConGolog with 3APL. ConGolog was conceived of as a language for high-level robot programming. ConGolog, like its predecessor Golog (Levesque et al. 1997), is an extension of the situation calculus that supports complex actions as well as a logic programming language for agents and robots. We show that ConGolog and 3APL are closely related languages by constructing an embedding of ConGolog in 3APL. A number of interesting issues need to be resolved to construct the embedding. These include a comparison of states in 3APL with situations in ConGolog, the form of basic action theories, complete vs. incomplete knowledge, and execution models specifying the flow of control in agent programs.

First, we introduce the situation calculus and a formalisation of action theories within this formalism. ConGolog is an extension of these action theories to a real programming language. The semantics of ConGolog is defined by means of the situation calculus. In the second half of the chapter, we define a translation function of ConGolog programs into 3APL agents and prove that the function defines a translation bisimulation.

10.1 Action Theories in the Situation Calculus

ConGolog is a programming language specified in and based upon the *situation calculus* (McCarthy & Hayes 1969). ConGolog extends basic action theories in the situation calculus to a real programming language. It allows the construction of composed program structures built from basic actions. Basic action theories are used to specify the preconditions and effects of basic actions that are used in ConGolog programs. A basic action theory only fixes the basic structure for specifying actions, but leaves the choice of basic actions to the programmer. We first introduce the situation calculus and then define what a basic action theory is.

10.1.1 The Situation Calculus

The situation calculus is a three-sorted, first order logical language, extended with some second order features. The situation calculus is specifically designed for representing dynamically changing worlds. Changes are the result of named, deterministic actions, and a possible world history therefore can be identified with a sequence of actions. Finite action histories are represented by first order terms called *situations* in the situation calculus. The language of the situation calculus $\mathcal{L}_{sitcalc}$ has three sorts: A sort *situation*, a sort *action*, and a sort *object* for everything that is neither a situation nor an action.

Definition 10.1.1 *(alphabet of $\mathcal{L}_{sitcalc}$)*
The alphabet of $\mathcal{L}_{sitcalc}$ consists of the following sets of symbols:

- Countably infinitely many variables for each sort; we use s to denote variables of sort *situation*, a for variables of sort *action*, and x, y for variables of sort *object*.
- Two function symbols of sort *situation*: (1) the constant S_0 denoting the *initial situation*, and (2) the function *do* of sort : *action* $\times$ *situation* $\rightarrow$ *situation* where $do(a, s)$ denotes the *successor situation* resulting from performing action a in situation s.
- A binary predicate $\sqsubset$ of sort : *situation* $\times$ *situation* which is defined as a partial order on situations.
- A binary predicate *Poss* of sort : *action* $\times$ *situation*. The intended interpretation of $Poss(a, s)$ is that a can be executed in s.
- A finite number of predicate symbols for each sort $(action \cup object)^n$ and function symbols for each sort $(action \cup object)^n \rightarrow (action \cup object)$. These predicate and function symbols are situation-independent.
- A finite number of predicate symbols of sort $(action \cup object)^n \times situation$. These predicate symbols are called *relational fluents*.

Note that only two function symbols - S_0 and *do* - are allowed to take values in sort *situation*. Also note that only the binary predicate $\sqsubset$ has more than one argument of sort *situation*. The language $\mathcal{L}_{sitcalc}$ is built from a given alphabet and the usual logical vocabulary, i.e. equality, negation, conjunction, and the universal quantifier. The other logical connectives like $\vee, \rightarrow, \leftrightarrow$ and the existential quantifier $\exists$ are defined as the usual abbreviations. $\leftrightarrow$ is also written as $\equiv$.

Notice that we did not include *functional fluents* in the language of the situation calculus. Functional fluents are left out because of the particular *call-by-value* mechanism that is used as a parameter mechanism in ConGolog for procedure calls, which would *not* be state based in the presence of functional fluents. For our purposes, we are only interested in eliminating functional fluents; other types of function symbols are allowed.

In the sequel, we will often be interested in the formulas that hold in the 'current' situation s, and that only refer to the situation s. To identify the formulas that talk about a particular situation, we introduce the notions of a *uniform term* and a *uniform formula.* A term or formula that is uniform *in a situation s* only refers to s.

Definition 10.1.2 *(uniform term, formula)*
Let S be any term of sort situation. Then the set T_S of terms *uniform in S* is inductively defined by:

- $S \in T_S$,
- if a term t does not mention a term of sort situation, then $t \in T_S$,
- if f is an n-ary function symbol other than do and $t_1, \ldots, t_n \in T_S$ whose sorts are appropriate for f, then $f(t_1, \ldots, t_n) \in T_S$.

The set $\mathcal{L}_S$ of formulas *uniform in S* is inductively defined by:

- if $t_1, t_2 \in T_S$ are of the same sort, then $t_1 = t_2 \in \mathcal{L}_S$,
- if P is an n-ary predicate symbol, other than $Poss$ and $\sqsubset$, and $t_1, \ldots, t_n \in T_S$ are of the appropriate sorts, then $P(t_1, \ldots, t_n) \in \mathcal{L}_S$
- if $\varphi_1, \varphi_2 \in \mathcal{L}_S$, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2 \in \mathcal{L}_S$
- if $\varphi \in \mathcal{L}_S$ and x is a variable not of sort situation, then $\forall\, x(\varphi) \in \mathcal{L}_S$.

A formula that is uniform in S does not mention the predicates $Poss$ or $\sqsubset$, nor does it quantify over situation variables. The only term of sort situation which can occur in a formula that is uniform in S is S itself. In a model M (and valuation ν) for $\mathcal{L}_{sitcalc}$, the true formulas which are uniform in S can be said to *characterise* situation S. In other words, these formulas *completely* specify the state denoted by S.

Notation 10.1.3 We introduce a special constant now of sort situation and denote by $\mathcal{L}_{now}$ the set of formulas uniform in now. The intended interpretation of this constant is that it denotes the current situation. If σ is any (set of) formula(s) that is uniform in now, we denote by $\sigma[S]$ the (set of) formula(s) that is obtained by substituting S for now in σ. Note that $\sigma[S]$ is uniform in S.

10.1.2 Foundational Axioms

The basic intuitions associated with the notion of a situation are captured by a set of so called *foundational axioms* (cf. Pirri & Reiter (1999)). These axioms are listed in definition 10.1.4. The first axiom below states that situations are uniquely identified by situation terms, and implies that situations can be identified with action histories. The second axiom is a second order axiom which captures the intuition that all the situations that exist are the ones reachable by doing a finite number of actions. The third axiom states that S_0 is the initial situation. And finally, the fourth axiom states that a situation s is a predecessor of a situation $do(a, s')$ iff s is a predecessor of s' or s and s' denote the same situation. Although these foundational axioms impose a basic structure upon the set of possible situations, the axioms do not play an important role in the definition of the programming language ConGolog (cf. also Pirri & Reiter (1999)).

Definition 10.1.4 *(foundational axioms)* [1]

$$do(a_1, s_1) = do(a_2, s_2) \rightarrow (a_1 = a_2 \wedge s_1 = s_2) \tag{10.1}$$

$$\forall P([P(S_0) \wedge \forall a, s(P(s) \rightarrow P(do(a, s))] \rightarrow \forall s(P(s)) \tag{10.2}$$

$$\neg s \sqsubset S_0 \tag{10.3}$$

$$s \sqsubset do(a, s') \equiv s \sqsubseteq s' \tag{10.4}$$

where $s \sqsubseteq s'$ abbreviates $s \sqsubset s' \vee s = s'$.

10.1.3 Basic Actions

A basic action theory in the situation calculus defines a framework for specifying the pre- and postconditions of actions. Three types of axioms are introduced to specify actions. First of all, a set of *unique names axioms for actions* is introduced. These axioms are used to make sure that action names refer to different actions, and that an action symbol supplied with one set of parameters is distinguished from that action symbol supplied with a different set of parameters.

Definition 10.1.5 *(unique names axioms for actions)*
The set of *unique names axioms for actions* includes the following axioms:

$$\mathsf{a}(\vec{x}) \neq \mathsf{b}(\vec{y})$$

where $\mathsf{a}(\vec{x})$ and $\mathsf{b}(\vec{y})$ are expressions of sort *action*, and the set of variables $\vec{x}$ and $\vec{y}$ are disjoint, for each pair of action symbols a and b;
and for any action symbol a:

$$\mathsf{a}(\vec{x}) = \mathsf{a}(\vec{y}) \rightarrow \vec{x} = \vec{y}.$$

[1]The free variables in formulas which occur in definitions throughout this chapter are implicitly universally quantified.

The second type of axioms are called *action precondition axioms.* These axioms specify when an action is *enabled*, i.e. they specify what preconditions must hold in order for an action to be executable in a situation. The uniformity condition on $\Pi_{\mathsf{a}}(\vec{x}, s)$ is used to make sure that the preconditions of an action $\mathsf{a}(\vec{t})$ depend only on the current situation s.

Definition 10.1.6 *(action precondition axiom)*
An *action precondition axiom* is of the form:

$$Poss(\mathsf{a}(\vec{x}), s) \equiv \Pi_{\mathsf{a}}(\vec{x}, s)$$

where $\mathsf{a}(\vec{x})$ is an expression of sort *action*, and $\Pi_{\mathsf{a}}(\vec{x}, s)$ is a formula that is uniform in s and whose free variables are among $\vec{x}, s$.

The last type of axioms are called *successor state axioms.* Successor state axioms relate the value of a fluent in the situation that results from doing an action to their value in the previous situation, and define the effects of executing an action. Successor state axioms also provide a solution to the frame problem (Reiter 1991). The uniformity condition on $\Phi_F(\vec{x}, a, s)$ guarantees that the database associated with the successor situation (the database of uniform formulas that hold in that situation) can be computed from that of the previous situation.

Definition 10.1.7 *(successor state axiom)*
A *successor state axiom* for a *relational* fluent F is of the form:

$$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$$

where $\Phi_F(\vec{x}, a, s)$ is a formula uniform in s and whose free variables are among $\vec{x}, a, s$.

A basic action theory is a collection of the axioms introduced so far. Our definition of a basic action theory slightly differs from the one in (Lin & Reiter 1997). The main difference is that we do not include the initial situation axioms or the *initial database* in the action theory.

Definition 10.1.8 *(basic action theory)*
A *basic action theory* is a theory $\mathcal{A} = \Sigma \cup \mathcal{A}_{ss} \cup \mathcal{A}_{ap} \cup \mathcal{A}_{una}$ where:

- Σ are the *foundational axioms*,
- $\mathcal{A}_{ss}$ is a set of *successor state axioms* for relational fluents, one for each fluent,
- $\mathcal{A}_{ap}$ is a set of *action precondition axioms*, one for each action symbol,
- $\mathcal{A}_{una}$ is a set of *unique names axioms*, for all pairs of action function symbols.

Definition 10.1.9 *(initial database)*
An *initial database* is a finite set of (first order) sentences from $\mathcal{L}_{sitcalc}$ that are uniform in S_0.

10.1.4 Situations, States and Functional Fluents

The reason for introducing the uniformity conditions into basic action theories is to ensure that the evaluation of preconditions and successor state conditions depends only on the current situation. In the presence of functional fluents, however, the uniformity conditions are not enough to guarantee that the preconditions and successor state conditions depend only on the current situation. It is not too difficult to give an example in which a functional fluent is substituted for a parameter and results in a violation of a uniformity condition. For example, if we substitute $loc(Ball, do(throw(Ball), S_0))$ for x and S_0 for s in $Poss(goto(x), s) \equiv reachable(x, s)$, we obtain the precondition $reachable(loc(Ball, do(throw(Ball), S_0)), S_0)$ which is *not* uniform in S_0. In the presence of functional fluents, therefore, we have to be more careful and only substitute terms that do not lead to violations of the uniformity conditions.

The fact that all conditions can be evaluated by inspection of the current situation only implies that during a computation only a database of facts that talk about the current situation has to be maintained. This is a typical feature of a *state based approach.* The main characteristic of a state based approach is that a successor state can be computed from the current state and the action that is performed in that state. Because of the particular form of successor state axioms, basic action theories also support a state based approach, with the proviso that substitution of functional fluents does not lead to violations of the uniformity conditions.

The uniformity conditions thus play an important role in basic action theories. In general, there is an important difference between situations in the situation calculus and states in state based approaches. Whereas a state coincides with a single point in (space-)time, a situation (action history) can be much more complex. A situation s can even refer to would-be situations in a possible history that is different from the actual one referred to by s. For example, consider the situation $do(goto(loc(Ball, do(trow(Ball), S_0)), S_0)$. This situation refers to the situation resulting from going to a particular place in the initial situation S_0. The place referred to needs to be inferred from doing *another* action in situation S_0, namely the action of throwing a ball. For the evaluation of a formula like $corner(loc(Robot, do(goto(loc(Ball, do(throw(Ball), S_0)), S_0))$ we thus have to inspect the would-be situation resulting from throwing the *Ball* in situation S_0, and check if the location of the *Ball* in *that* situation is a corner, assuming that a goto action always succeeds. Due to the possibility of a branching structure of situations we can construct such 'non-linear' situations which depend on other situations in different branches in the possible histories structure.

The example of the previous paragraph used functional fluents to illustrate that situations are different from states. Still another feature in the situation calculus, that of quantification over situations, can give rise to formulas that refer to different situations. An example of such a formula is the following precondition axiom: $Poss(open(d), s) \equiv \exists s'(s = do(unlock(d, key), do(get(key), s')))$. This formula states that it is only possible to open a door if a key has been obtained and the door is unlocked with this key in the last *two* situations. Because this precondition refers to the two previous situations, it cannot be evaluated by inspecting the current situation only.

Summarising, in general the situation calculus offers an expressive framework for talking about action histories. The basic action theories that we introduced restrict this expressivity by introducing uniformity conditions, and thus provide for a state based approach. Still, we have to be careful in the presence of functional fluents. Because 3APL is a state based formalism, for the purpose of bisimulating ConGolog, it is important that basic action theories are state based. This is our main reason for excluding functional fluents.

10.2 The Programming Language ConGolog

ConGolog is a programming language based on the situation calculus. It extends the basic action theories of the previous section with operators for constructing complex actions. In ConGolog it is possible, for example, to specify the sequential composition of two actions, like, $\mathit{pickup}(\mathit{Block})$; $\mathit{putaway}(\mathit{Block})$. The set of ConGolog programs is defined below. It is a subset of all the programs as in (Giacomo et al. 2000), but includes the main programming constructs. Most of the programming constructs below are well-known. Tests evaluate a formula in the current situation. The nondeterministic choice of argument construct nondeterministically selects a value for the variable x. In a prioritised parallel program $\delta_1 \rangle\!\rangle \delta_2$ the execution of the left subprogram δ_1 is preferred over that of the right subprogram δ_2; the latter is executed only if δ_1 cannot be executed.

Definition 10.2.1 *(ConGolog programs)*
The set of *open* programs P and procedures *Proc* is inductively defined by:

- *primitive actions:* $\mathsf{a}(\vec{t}) \in P$, for $\mathsf{a}(\vec{t})$ of sort action,
- *tests:* $\phi? \in P$, for $\phi \in \mathcal{L}_{now}$,
- *sequential composition:* $(\delta_1;\ \delta_2) \in P$, if $\delta_1, \delta_2 \in P$,
- *nondeterministic choice:* $(\delta_1 \mid \delta_2) \in P$, if $\delta_1, \delta_2 \in P$,
- *nondeterministic choice of arguments:*
  $\pi x.\delta \in P$, if $\delta \in P$ and x is a variable of sort object, [2]
- *parallel composition:* $\delta_1 || \delta_2 \in P$, if $\delta_1, \delta_2 \in P$,
- *prioritised parallel composition:* $\delta_1 \rangle\!\rangle \delta_2 \in P$, if $\delta_1, \delta_2 \in P$,
- *procedure call:* $P(\vec{t})$,
- *procedure definition:*
  proc $P(\vec{x})$ δ_P **end** $\in$ *Proc*, if $\delta_P \in P$ and all variables in δ_P occur in $\vec{x}$.

[2]Variables of sort action can be simulated if there are only a finite number of actions available. An example of the use of action variables is given in (Giacomo et al. 2000).

By definition, the set of *ConGolog programs* is the set of *closed* programs in P.

A number of constructs that are included in ConGolog are not included in the previous definition. The constructs which are defined in (Giacomo et al. 2000) but are not included in the definition above are iteration, synchronised if-then-else, synchronised while, and parallel iteration. As far as iteration is concerned, no expressivity is lost, since it is well-known that this construct can be simulated by recursive procedures which are included in definition 10.2.1. The synchronised if-then-else and the synchronised while are slight variations of the non-synchronised ones. Both of these constructs require that the test as well as the first action of one of the branches of the if-then-else or of the body of the while-construct are executed in a single step. Thus in both constructs the test and the first action to be executed next are considered to be an atomic step that cannot be interrupted. 3APL does not have similar constructs that are synchronised in this way. However, it would not be difficult to extend 3APL with similar constructs, but doing so would not lead to any new or interesting results with respect to the simulation of ConGolog in 3APL. A similar remark applies to parallel or concurrent iteration.

10.2.1 Axiomatic Definition of the Semantics for ConGolog

The meaning of the ConGolog programming constructs is specified by using a transition semantics that is presented in a non-standard way. Instead of using a formalism like the transition systems used in structural operational semantics (Plotkin 1981), a new predicate *Trans* is added to the language of the situation calculus and is used to formalise the step semantics of a program. The predicate $\mathit{Trans}(\delta, s, \delta', s')$ expresses that it is possible for program δ to perform a computation step in situation s that results in a new situation s' where δ' is the remaining program that still needs to be executed. The semantics of ConGolog programs is specified by means of a set of axioms for the predicate *Trans*. For each programming construct, there is an axiom that states which computation steps the construct allows. The expression *nil* denotes the 'empty' program, and is used below as an auxiliary construct in the definition of the operational semantics. *nil* is *not* a ConGolog program. Also notice that no transition is associated with the nondeterministic selection of a value in a $\pi x.\delta$ program, but only with the joined action of selecting a value and executing a step of subprogram δ. In the axiomatic definition of *Trans*, the predicate $\mathit{Final}(\delta, s)$ is used to express that program δ may legally terminate in situation s. A formal definition of *Final* is presented after the definition of *Trans*. The definition of the semantics of (recursive) procedures is postponed until the next section.

Definition 10.2.2 *(axioms for Trans)* [3]
Trans is inductively defined by:

- The Empty Program:

$$Trans(nil, s, \delta', s') \equiv False$$

- Basic Actions:

$$Trans(a, s, \delta', s') \equiv Poss(a, s) \wedge \delta' = nil \wedge s' = do(a, s)$$

- Tests:

$$Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$$

- Sequential Composition:

$$\begin{aligned} & Trans(\delta_1;\ \delta_2, s, \delta', s') \equiv \\ & \quad (\exists\, \gamma.\delta' = (\gamma;\ \delta_2) \wedge Trans(\delta_1, s, \gamma, s')) \vee \\ & \quad (Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')) \end{aligned}$$

- Nondeterministic Choice:

$$Trans(\delta_1 \mid \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$$

- Nondeterministic Choice of Argument:

$$Trans(\pi x.\delta, s, \delta', s') \equiv \exists\, x. Trans(\delta, s, \delta', s')$$

- Parallel Composition:

$$\begin{aligned} & Trans(\delta_1 \| \delta_2, s, \delta', s') \equiv \\ & \quad \exists\, \gamma.\delta' = (\gamma \| \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \\ & \quad \exists\, \gamma.\delta' = (\delta_1 \| \gamma) \wedge Trans(\delta_2, s, \gamma, s') \end{aligned}$$

- Prioritised Parallel Composition:

$$\begin{aligned} & Trans(\delta_1 \rangle\!\rangle \delta_2, s, \delta', s') \equiv \\ & \quad \exists\, \gamma.\delta' = (\gamma \rangle\!\rangle \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \\ & \quad \exists\, \gamma.\delta' = (\delta_1 \rangle\!\rangle \gamma) \wedge Trans(\delta_2, s, \gamma, s') \wedge \neg \exists\, \eta, s''. Trans(\delta_1, s, \eta, s'') \end{aligned}$$

[3]Formally, an encoding of ConGolog programs into terms of the first order language $\mathcal{L}_{sitcalc}$ is required, as is done in (Giacomo et al. 2000). However, because the details in this chapter - apart from the encoding itself - are almost completely the same, for notational convenience we use the programs as in definition 10.2.1 and refer the reader to (Giacomo et al. 2000) for the details concerning the encoding of programs into terms.

Definition 10.2.3 *(axioms for Final)*
The *Final* predicate is defined by the following set of axioms:

- The Empty Program:

$$Final(nil, s) \equiv True$$

- Basic Actions:

$$Final(a, s) \equiv False$$

- Tests:

$$Final(\phi?, s) \equiv False$$

- Sequential Composition:

$$Final(\delta_1;\ \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

- Nondeterministic Choice:

$$Final(\delta_1 \mid \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$$

- Nondeterministic Choice of Argument:

$$Final(\pi x.\delta, s) \equiv \exists\, x.Final(\delta, s)$$

- Parallel Composition:

$$Final(\delta_1 \| \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

- Prioritised Parallel Composition:

$$Final(\delta_1 \rangle\rangle \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$$

10.2.2 ConGolog Procedures

The semantics of ConGolog procedures is not defined in terms of replacement of the procedure *call* with the procedure *body*, since such steps are not viewed as transitions in the ConGolog semantics (cf. Giacomo et al. (2000)). Instead, a second order definition of the transition predicate is given which abstracts from these steps. A procedure call in the ConGolog semantics involves both body replacement of a (number of) procedure call(s) and the execution of an action or test in a single step. Only actions or tests can give rise to transitions in the ConGolog semantics, and replacement of a procedure call with its associated body or the nondeterministic selection of a value in a $\pi x.\delta$ program are not viewed as transitions.

In ConGolog, *nesting of procedure definitions* is allowed. Therefore, it is important to keep track of the scope of a procedure definition. Nesting of procedure definitions, however, can be considered as syntactic sugar. A pre-compiler can be used to remove all naming conflicts and in that case we may assume global scope again. we do not consider this facility here. The implementation of ConGolog also does not include this feature (cf. Giacomo et al. (2000)). In the absence of procedure nestings, the second order definition of *Trans* that also deals with procedures can be defined by

$$Trans(\delta, s, \delta', s') \equiv \forall\, T.(\varphi(T, \delta, s, \delta', s') \rightarrow T(\delta, s, \delta', s'))$$

where $\varphi(T, \delta, s, \delta', s')$ is the conjunction of the set of axioms for *Trans* of the previous section, with T substituted for *Trans*, and the following clause for procedure calls:

$$T(P(\vec{t}), s, \delta', s') \equiv T((\delta_P)^{\vec{x}}_{\vec{t}}, s, \delta', s').$$

In the clause for procedure calls, δ_P is the body of the procedure definition of $P(\vec{x})$ and $(\delta_P)^{\vec{x}}_{\vec{t}}$ is that same body where the formal parameters $\vec{x}$ have been substituted with $\vec{t}$. This second order definition defines *Trans* as the smallest set of transitions closed under the set of clauses for the ConGolog programming constructs defined in the previous section and the clause for procedure calls introduced in this section.

The predicate *Final* is defined in a similar way as the *Transe* predicate:

$$Final(\delta, s) \equiv \forall\, F.(\psi(F, \delta, s) \rightarrow F(\delta, s))$$

where $\psi(F, \delta, s)$ is the conjunction of the set of axioms for *Final* of the previous section, with F substituted for *Final*, and the following clause for procedure calls:

$$F(P(\vec{t}), s) \equiv F((\delta_P)^{\vec{x}}_{\vec{t}}, s).$$

The parameter mechanism of ConGolog is a *call-by-value* mechanism, which, due to the assumption that functional fluents are absent, could be somewhat simplified compared to the definition in (Giacomo et al. 2000). In the presence of functional fluents, the substitution of the actual parameters is slightly more complex and should be $(\delta_P)^{\vec{x}}_{\vec{t}[s]}$ instead of simply $(\delta_P)^{\vec{x}}_{\vec{t}}$. That is, functional fluents in an actual parameter term t should be evaluated with respect to the current situation s, and to implement this the situation s is substituted for the constant now. After substituting s for now in parameter t, the parameter $t[s]$ is substituted in the body of the procedure at the appropriated places.

We are now in a position to explain in detail our remark above that this type of parameter mechanism may result in an approach that is not state based, if functional fluents are allowed. Consider, for example, the procedure definition **proc** *serve*(n) *go_floor*(n); *turnoff*(n); *open* **end** and the procedure call *serve*$($*nearest_floor*$($*now*$))$. Notice that the actual parameter in the procedure call is a functional fluent. Now suppose that the current situation is the initial situation S_0. In that case, we get the following transition as a result of the execution of the call:

$$\begin{aligned}&\mathit{Trans}(\mathit{serve}(\mathit{nearest\_floor}(\mathit{now})), S_0,\\&\quad\mathit{turnoff}(\mathit{nearest\_floor}(S_0));\ \mathit{open}, \mathit{do}(\mathit{go\_floor}(\mathit{nearest\_floor}(S_0)), S_0))\end{aligned}$$

The remaining program is $\mathit{turnoff}(\mathit{nearest\_floor}(S_0));\ \mathit{open}$ which must be executed in situation $\mathit{do}(\mathit{go\_floor}(\mathit{nearest\_floor}(S_0)), S_0)$. For the evaluation of the argument $\mathit{nearest\_floor}(S_0)$, however, the program has to consult the *previous* situation S_0 instead of the current one. The call-by-value mechanism thus is not compatible with a strictly state based approach in the presence of functional fluents.

A second difference of the ConGolog semantics of procedures and the 3APL semantics for achievement goals is that the second order semantics for ConGolog procedures abstracts from body replacement. That is, a step in which a procedure call is replaced with its body is not considered to be a separate transition. Only actions and tests are viewed as transitions, and, as a consequence, a procedure call $P(\vec{t})$ that never gives rise to the execution of an action or test, can never give rise to any transition. Formally, $\forall \delta', s'. \neg \mathit{Trans}(P(\vec{t}), s, \delta', s')$. This is quite different from a semantics that associates a transition with body replacement as is done in the semantics for 3APL. Consider, for example, the procedure definition

$$\textbf{proc}\ d(n)\ (n = 1)? \mid d(n-1);\ \mathit{go\_down}\ \textbf{end}$$

and the program $d(0) \mid \mathsf{true}?$. According to the ConGolog semantics, the only transition this program can (always) make is the transition in which the test is executed, because the procedure call $d(0)$ never gives rise to the execution of an action or test. This program thus always successfully executes and terminates after executing the test in the ConGolog semantics. In the 3APL semantics, this is not the case. Since body replacement also is a legal computation step, the left branch may be selected and a body replacement may occur. The selection of the left branch, however, results in a non-terminating computation where in each step a body replacement is performed. In the 3APL semantics, the program thus has a non-terminating computation in contrast with the ConGolog semantics.

Although the behaviour in accordance with the ConGolog semantics may be preferred over that of a semantics which includes steps for body replacement that gives rise to a non-terminating computation, there is a computational problem. To implement the second order semantics for procedure calls, an algorithm which decides if such a call results in the eventual execution of an action or test is required. The problem is that such an algorithm does not exist, since that algorithm would also solve the halting problem. In our example, non-termination may seem easy to detect, but in general it is not possible to decide this type of termination for arbitrary actual parameters (which may involve complex terms). The extension of an operational style semantics in first order logic with second order axioms in this case thus results in a non-computational semantics.

Due to the fact that ConGolog and 3APL assign different semantics to procedure calls, it is not possible to construct an embedding of ConGolog into 3APL. An embedding, however, can be constructed for a large and interesting subclass of ConGolog procedure definitions. In particular, the set of *guarded procedures* is a class of procedures that never does more than a fixed number of procedure calls before executing an action or test. This class thus avoids the problem of detecting termination as in the general case. This subset also can be embedded into 3APL.

A formal definition of a *guarded program* is provided by means of the notion of a *rank*. This notion is similar to that in (Giacomo et al. 2000), except for one important difference. In contrast to the definition in (Giacomo et al. 2000), our notion of rank is *not* dependent on the current situation. It is completely syntactic and therefore somewhat simpler.[4]

Definition 10.2.4 *(rank)*
The *rank* n (n a natural number) of a program δ (possibly containing free variables) is defined by the following axioms:

$$\begin{array}{rcl}
Rank(n, nil) & \equiv & True \\
Rank(n, a) & \equiv & True \\
Rank(n, \phi?) & \equiv & True \\
Rank(n, \delta_1;\ \delta_2) & \equiv & Rank(n, \delta_1) \wedge Rank(n, \delta_2) \\
Rank(n, \delta_1 \mid \delta_2) & \equiv & Rank(n, \delta_1) \wedge Rank(n, \delta_2) \\
Rank(n, \pi x.\delta) & \equiv & Rank(n, \delta) \\
Rank(n, \delta_1 \| \delta_2) & \equiv & Rank(n, \delta_1) \wedge Rank(n, \delta_2) \\
Rank(n, \delta_1 \rangle\!\rangle \delta_2) & \equiv & Rank(n, \delta_1) \wedge Rank(n, \delta_2) \\
Rank(n, P(\vec{t})) & \equiv & Rank(n-1, \delta_P{}^{\vec{x}}_{\vec{t}})
\end{array}$$

A ConGolog program δ is *guarded*, denoted by $Guarded(\delta)$ iff δ is of rank n for some n, that is: $Guarded(\delta) \overset{df}{=} \exists\, n.Rank(n, \delta)$. This notion of guardedness is strictly stronger than the one defined in (Giacomo et al. 2000), because it does not depend on the current situation. As a consequence, Theorem 6 in (Giacomo et al. 2000) holds, and we can define the semantics of procedures by a set of first order axioms. In the sequel, we will prove an embedding result for *guarded programs* which is based on the first order semantics of the previous section and that of definition 10.2.5 below. It should be understood that the semantics of definition 10.2.5 only applies to guarded programs (procedure definitions).

Definition 10.2.5 *(first order axioms for guarded procedure calls)*

$$Trans(P(\vec{t}), s, \delta', s') \equiv Trans(\delta_P{}^{\vec{x}}_{\vec{t}}, s, \delta', s')$$
$$Final(P(\vec{t}), s) \equiv Final(\delta_P{}^{\vec{x}}_{\vec{t}}, s)$$

[4]As a consequence, if a procedure call is guarded for all procedures in a program δ, we also know that a program has a 'guarded evolution'; that is, any program resulting from the execution of any number of steps of δ is guarded again. This is our analogue of Theorem 7 in (Giacomo et al. 2000).

10.3 Simulating ConGolog with 3APL

Although at first sight, ConGolog and 3APL may seem quite different languages, the languages are similar in many respects. A short comparison of the main features quickly reveals the similarities and differences. Since our aim is to construct an embedding of ConGolog in 3APL, we have to find ways to deal with the differences between the languages. In the remainder of this chapter, we will discuss the differences between the languages and argue for particular ways to resolve them.

The knowledge representation of ConGolog and 3APL is quite similar. Both languages use a logical language. An initial database is used in ConGolog to specify initial information about an environment, whereas a belief base is used in 3APL. The beliefs of a 3APL agent are drawn from some knowledge representation language $\mathcal{L}$. In principle, the choice of knowledge representation language is free, and any formalism which allows the derivation of facts from the agent's beliefs can be used to program agents. For the purpose of simulating ConGolog, it is convenient to identify the knowledge representation language $\mathcal{L}$ with the language $\mathcal{L}_{now}$ of situation calculus formulas uniform in *now*. Subsets of these formulas are used to represent the current situation.

The goals of a 3APL agent are plans, or imperative programs like in ConGolog. These goals are built from basic actions, tests, and the same programming constructs as in ConGolog, apart from some minor differences in notation. Achievement goals correspond to procedure calls of ConGolog.

Originally, 3APL does not include a construct for prioritised parallel composition $\rangle\!\rangle$ (Hindriks et al. 1998, Hindriks et al. 1999*a*), but in the sequel we show how to formally define a semantics for $\rangle\!\rangle$ in a transition style semantics, and extend 3APL with this operator.

One of the more important differences between ConGolog and 3APL is the presence of the π operator in ConGolog and the absence of such a construct in 3APL. Correspondingly, the two languages have quite different parameter mechanisms. Whereas in ConGolog the π operator is used to nondeterministically select a value for a variable that is *bound* by the operator, in 3APL tests are used to compute values for *free* variables as in logic programming. Moreover, whereas the π operator provides for an explicit scoping mechanism, the use of free variables in 3APL is based on implicit scoping and involves the renaming of free variables when procedures calls are replaced with their corresponding body. To facilitate the construction of an embedding of ConGolog into 3APL and to accommodate for these different styles of parameter passing, we introduce an additional construct $random(x)$ which, like the πx operator, nondeterministically selects a value for the variable x. $random(x)$ does not introduce any additional expressivity into 3APL and can be viewed as syntactic sugar.

A $random(x)$ action, like a test, computes a binding for the variable x. In contrast with arbitrary tests, however, $random(x)$ always succeeds and nondeterministically returns an arbitrary binding for x. The $random(x)$ action does not increase the expressivity of 3APL, since it can be defined as a special kind of test. For an arbitrary unary predicate symbol p, $random(x)$ can be defined as: $random(x) \overset{df}{=} (p(x) \vee \neg p(x))?$. The reason for introducing $random(x)$ as an explicit action is that we want to label this particular action as a silent step. $random(x)$ is used to simulate the pick operator π, which does not give rise to a transition. The nondeterministic selection of a value thus is considered as an implementation detail in the ConGolog semantics, and therefore needs to be modelled as a silent step in the 3APL semantics. As before, silent steps are denoted by the label i.

Definition 10.3.1 *(transition rule for random)*
Let t be a ground term.

$$\frac{x \text{ is a variable}}{\langle random(x), \sigma\rangle_V \xrightarrow{i}_{\{x=t\}} \langle E, \sigma\rangle} \qquad \frac{}{\langle random(t), \sigma\rangle_V \xrightarrow{i}_{\varnothing} \langle E, \sigma\rangle}$$

Finally, a practical reasoning rule in 3APL has the form $\pi \leftarrow \phi \mid \pi'$, where π, π' are goals and ϕ is a formula from $\mathcal{L}_{now}$, the knowledge representation language we use here. If the head π is specialised to an achievement goal $p(\vec{x})$ and the guard is identified with true, then we obtain rules which correspond to the recursive procedures of ConGolog. For our purposes, we only need simple rules of the form $p(\vec{x}) \leftarrow \pi$ (a guard that is equivalent with true is not mentioned).

10.3.1 A Labelling of the Transition Relation

For the simulation of ConGolog, it is important to keep track of the *sequence of basic actions* which are executed during a computation of a program. This information is explicitly represented by the situation arguments of the *Trans* predicate in ConGolog. To represent the same information in the 3APL semantics, we define a *labelled transition relation.* Labels are associated with a transition and indicate whether an action or something else has been performed.

Labels are also used to distinguish between two types of transitions in 3APL. From the point of view of the ConGolog semantics only the execution of basic actions or tests give rise to a transition. As we will see below, 3APL associates a transition with the expansion of a procedure call into its body and with the execution of a *random* action for nondeterministically selecting values. We also use labels to distinguish between 3APL transitions that count as transitions in the ConGolog semantics too and the ones that do not count as such. The latter type of transitions are singled out as *silent steps.* The intuition is that these steps are not 'visible' in the ConGolog semantics and from the ConGolog perspective are considered to be implementation details.

The labelling is derived from these intuitions. A transitions that corresponds with the execution of a basic action is labelled with that same basic action. A transition that corresponds with the execution of a test is labelled with the special symbol ϵ, the empty sequence. The empty sequence is used to denote that no action has been performed, and the situation has not been changed. Both the execution of a *random* action and the expansion of a procedure call into its body are labelled with i to indicate that a silent step has been performed.

The labelling of composed programs are derived from the more basic ones. For example, the label associated with the execution of a sequential composition $\pi_1;\ \pi_2$ is derived from the label that is associated with the execution of π_1.

10.3.2 Observables and Silent Steps

As we explained above, in the semantics of ConGolog a transition is associated with a program only if it can perform an action or test. In 3APL, however, a transition is associated also with the expansion of a call into its body and with the execution of a *random* action. For this reason, we introduced a distinction between two types of computation steps in 3APL. Computation steps due to the execution of a basic action or a test are distinguished from other types of computation steps. The latter are singled out as *silent steps.*

For the construction of an embedding of ConGolog in 3APL, we want to abstract from these silent steps. For this reason, we introduce a new transition relation $\Longrightarrow$ for 3APL programs that is derived from the transition relation $\longrightarrow$ defined in chapter 3. The transition relation $\Longrightarrow$ induces a new step relation. $\Longrightarrow$ steps are composed of an arbitrary number of silent steps $\xrightarrow{i}$ followed by a single step that involves the execution of a basic action or a test.

In the definition below, * denotes the transitive closure of a relation. The relation $\longrightarrow_\gamma^*$ is defined as the set of all finite (including empty) sequences of $\longrightarrow$ steps and γ is defined as the subsequent application of the substitutions associated with each of these steps. More formally, we have that for *all* γ and *all* n: if $\longrightarrow_{\gamma_1}\longrightarrow_{\gamma_2} \dots \longrightarrow_{\gamma_n}$ denotes a legal sequence of steps such that $\gamma = \gamma_1\gamma_2\dots\gamma_n$, then $\longrightarrow_{\gamma_1}\longrightarrow_{\gamma_2} \dots \longrightarrow_{\gamma_n} \in \longrightarrow_\gamma^*$.

Definition 10.3.2 *(abstracting from silent steps)*
The transition relation $\overset{l}{\Longrightarrow}$, where l is either $\mathsf{a}(\vec{t})$ or ϵ, is defined by:

$$\begin{array}{rcl} \overset{\mathsf{a}(\vec{t})}{\Longrightarrow}_\gamma & \overset{df}{=} & \overset{i}{\longrightarrow}_\gamma^{*} \cdot \overset{\mathsf{a}(\vec{t})}{\longrightarrow}_\varnothing \\ \overset{\epsilon}{\Longrightarrow}_{\gamma\theta} & \overset{df}{=} & \overset{i}{\longrightarrow}_\gamma^{*} \cdot \overset{\epsilon}{\longrightarrow}_\theta \end{array}$$

In the sequel, we also just write $\Longrightarrow$ or $\longrightarrow$ instead of $\Longrightarrow_\gamma$ or $\longrightarrow_\gamma$ in case substitutions are not important in the context.

The derived transition relation $\Longrightarrow$ imposes an order on the steps performed by a 3APL agent. This order does not impose strong restrictions on the execution of an agent. It requires that an agent only performs a step if it is going to perform an action or a test sooner or later. Since procedures are assumed to be guarded, any translation of a ConGolog program satisfies this requirement. It is with respect to the new step relation $\Longrightarrow$ that we are going to prove that ConGolog can be embedded in 3APL.

The observables that are used in the embedding include both state based as well as action-based observables. The action-based observables are used to keep track of the actions that have been performed. The belief base of an agent is again taken as the state based observable. (The database of a basic action theory that is associated with an agent provides the information state of a ConGolog program.) The labels associated with the $\Longrightarrow$ relation are taken as the action-based observables. The same information can be retrieved from the situation argument in the situation calculus.

10.4 Operationalising ConGolog

In this section, we discuss a number of distinguishing features of the ConGolog and 3APL semantics. To be able to construct an embedding of ConGolog into 3APL, a number of issues have to be dealt with. First, we discuss the semantics of tests. From this discussion, we derive a requirement on the initial database or belief base of an agent. Secondly, we discuss the semantics of nondeterministic selection of a value by the π operator. We conclude that a domain closure - or similar - assumption is needed to operationalise this operator. These discussions show a difference between ConGolog and 3APL due to the different formalisms used to define their respective semantics. The ConGolog semantics offers a logical definition of an agent system, whereas the 3APL semantics offers a more operational or computational definition of agent systems. The logical semantics of ConGolog raises a number of issues as to how to operationalise or implement it. Next, we proceed to show how to derive an update semantics for 3APL actions from the successor state axioms provided by a basic action theory $\mathcal{A}$. And finally, we define a translation function τ that maps ConGolog programs to equivalent 3APL goals. In section 10.5, we then prove that the translation function τ defines an embedding of ConGolog into 3APL.

10.4.1 Operationalising Tests and Complete Theories

A particularly interesting difference between ConGolog and 3APL concerns the semantics of tests. Whereas the semantics of a test in 3APL is defined in terms of entailment by the current beliefs, in ConGolog a test is defined in terms of truth in the current situation. The difference can be illustrated with the program $\delta = (\phi?;\ \mathsf{a}) \mid (\neg\phi?;\ \mathsf{a})$. In 3APL, δ is not always enabled because it is possible that neither ϕ nor its negation $\neg\phi$ is entailed by the current beliefs of the agent. In contrast, δ is enabled in any situation s in ConGolog and, if a is also enabled in situation s, results in a final situation $do(\mathsf{a}, s)$. δ is enabled simply because either ϕ or $\neg\phi$ must hold in the current situation (we may assume that ϕ is closed since ConGolog programs are closed).

Program δ of the previous paragraph is a simple example that illustrates the difference between ConGolog and 3APL tests. The program δ, however, is a special kind of program since it does not raise the question whether or not the left or right branch of the program should be executed. Either way, action a must be executed. In case we replace a in both branches with different subprograms, however, we do need to determine which branch to execute. Consider the program $\delta' = \phi?;\ \mathsf{a} \mid \neg\phi?;\ \mathsf{b}$. The ConGolog semantics implies that δ' is enabled in any situation, and one of the tests in one of the branches should be executed. Due to *incomplete information* about the current situation, however, it may be impossible to *decide* which branch needs to be executed.

To give still another example, in case the initial database does not contain any information about the proposition P, the ConGolog semantics does not specify the behaviour of program $P?$. The logical semantics allows models in which $P?$ can be executed and the program terminates successfully, and models in which $P?$ is not enabled and the program never terminates successfully. In contrast, the 3APL semantics in such a case precisely specifies the behaviour of the program since we always have $\sigma \models P$ or $\sigma \not\models P$ for arbitrary belief bases σ and propositions P.

The ConGolog semantics of tests thus raises the issue how to operationalise or implement it, since, as is illustrated by the examples, the ConGolog semantics does not always completely specify the behaviour of a program with tests. A possible solution for this problem is to require that (initial) databases are *complete*. A complete database σ decides every sentence of a language, and as a consequence if $\sigma \models \phi \vee \psi$ we also have that $\sigma \models \phi$ or $\sigma \models \psi$ for any ϕ, ψ. As a consequence, the two decision problems, evaluating whether ϕ or $\neg\phi$ holds in the current situation, or $\sigma \models \phi$ or $\sigma \models \neg\phi$, coincide. In our case, it is essential to be able to evaluate arbitrary sentences uniform in a particular situation and therefore we require that the current database associated with a particular situation is complete.

Definition 10.4.1 *(complete theories)*
Let $\sigma \subseteq \mathcal{L}$. σ is called *complete* iff for every sentence φ in $\mathcal{L}$ either $\sigma \models \varphi$ or $\sigma \models \neg\varphi$.

Lemma 10.4.2 Let $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory, $\varphi \in \mathcal{L}_{now}$, and S be a closed situation term. Then, for any action theory $\mathcal{A}$,

$$\mathcal{A} + \sigma[S] \models \varphi[S] \text{ iff } \sigma \models \varphi$$

Proof: Immediate, since $\varphi[S]$ is uniform in S and σ is a complete theory. □

10.4.2 Operationalising the π Operator and Domain Closure

Another interesting difference concerns the parameter mechanism in ConGolog and 3APL. Whereas ConGolog has an explicit operator πx which binds variables in a program (only closed programs are ConGolog programs!), the parameter mechanism in 3APL is based on an implicit binding mechanism and the use of tests for computing values for *free* variables in a goal.

The axiomatic definition of the π operator by means of the *logical existential quantifier*, however, again raises the issue of how to operationalise or implement the operator. In the 3APL semantics, computing bindings for (free) variables is specified as finding a suitable *term* to instantiate the variable. The logical semantics for the π operator, however, does not specify any particular mechanism for implementing it. Presumably, any implementation will have to manipulate terms and a similar mechanism as that of 3APL (based on a logic programming like parameter mechanism) is required.

To illustrate the difference between ConGolog and 3APL, we give a simple example. Suppose the language $\mathcal{L}_{now}$ (the knowledge representation language) only has a single constant a and no other function symbols. Furthermore, assume that the initial database is $\neg P(a)$. Now consider the program $\pi x.P(x)?$ and the question whether this program has a successfully terminating computation. As we will see, the 3APL translation of this ConGolog program is $\pi = \mathit{random}(x);\ P(x)?$ and it is easy to show that in this example the program π has no successfully terminating computation. In contrast, according to the ConGolog semantics the program has a successfully terminating computation in case $\exists\, x.P(x)$, which we cannot exclude given that we only know that $\neg P(a)$ is the case. The point is that there are models which satisfy $\neg P(a)$ and $\exists\, x.P(x)$ for some value in the domain, but this value has no *name*. The logical semantics of ConGolog thus constrains the behaviour of programs less than the 3APL semantics at the cost of not being able to prove certain useful properties concerning, for example, the termination behaviour of a program.

An elegant proposal to operationalise the nondeterministic selection of a value by the π operator is to assume *domain closure*. Domain closure implies that all domain elements have names, which provides for a computational mechanism to implement the π operator by computing bindings. From now on, therefore, we assume that action theories imply domain closure. That is, an action theory $\mathcal{A}$ now also includes a *domain closure axiom* like $\forall\, x(x = t_1 \vee \ldots x = t_n)$.

10.4.3 Some Useful Consequences

The fact that databases are required to be complete and that domain closure is assumed has a number of useful consequences. The most important ones are listed in this section and are used in the remainder of the chapter.

Lemma 10.4.3 Let $\mathcal{A}$ be a basic action theory and $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory. Then $\mathit{Final}(\delta, S)$ is decided by $\mathcal{A}{+}\sigma[S]$ for arbitrary ConGolog programs δ and closed situations S. That is,

$$\begin{array}{l}\mathcal{A} + \sigma[S] \models \mathit{Final}(\delta, S) \text{ or} \\ \mathcal{A} + \sigma[S] \models \neg \mathit{Final}(\delta, S)\end{array}$$

Proof: Easy induction on the structure of δ. Use domain closure for the case that δ is a nondeterministic choice of argument program. □

Theorem 10.4.4 Let $\mathcal{A}$ be a basic action theory and $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory. Then $\exists\, \delta', s'.\mathit{Trans}(\delta, S, \delta', s')$ is decided by $\mathcal{A} + \sigma[S]$ for arbitrary ConGolog programs δ and closed situations S. That is,

$$\begin{array}{l}\mathcal{A} + \sigma[S] \models \exists\, \delta', s'.\mathit{Trans}(\delta, S, \delta', s') \text{ or} \\ \mathcal{A} + \sigma[S] \models \neg \exists\, \delta', s'.\mathit{Trans}(\delta, S, \delta', s')\end{array}$$

Proof: By induction on the rank and structure of ConGolog programs. Induction on the rank of a program is used to deal with the case of Procedure Calls. This case is proven by a simple application of the induction hypothesis. The remaining cases are dealt with below.

- Basic Actions $\mathsf{a}(\vec{t})$: Follows from the fact that $\mathcal{A} + \sigma[S] \models \mathit{Poss}(\mathsf{a}(\vec{t}), S)$ or $\mathcal{A} + \sigma[S] \models \neg \mathit{Poss}(\mathsf{a}(\vec{t}), S)$. The latter is implied by the specific form of precondition axioms, the fact that σ is a complete theory, and lemma 10.4.2.
- Tests $\phi?$: Follows from the fact (lemma 10.4.2) that any sentence uniform in S is decided by $\mathcal{A} + \sigma[S]$.
- Sequential Composition $\delta_1; \delta_2$: By lemma 10.4.3, $\mathit{Final}(\delta_1, S)$ is decided. Then apply the induction hypothesis.
- Nondeterministic Choice $\delta_1 \mid \delta_2$: Use induction hypothesis.
- Nondeterministic Choice of Argument $\pi x.\delta$: Use domain closure and induction hypothesis.
- Parallel Composition $\delta_1 \| \delta_2$: Use induction hypothesis.
- Prioritised Parallel Composition $\delta_1 \rangle\!\rangle \delta_2$: Use induction hypothesis.

□

For an arbitrary term S of sort situation, as a notational shorthand, we stipulate that $do(\epsilon, S)$ is identical to S (where ϵ denotes the empty sequence; recall that ϵ is the label associated with the transition for a test in 3APL).

Theorem 10.4.5 Let $\mathcal{A}$ be a basic action theory, $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory and α be either ϵ or a basic action. Then any closed sentence of the form $\mathit{Trans}(\delta, S, \delta', do(\alpha, S))$ is decided by $\mathcal{A} + \sigma[S]$; that is, either:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta, S, \delta', do(\alpha, S)) \text{ or}$$
$$\mathcal{A} + \sigma[S] \models \neg \mathit{Trans}(\delta, S, \delta', do(\alpha, S))$$

Proof: By induction on the rank and structure of ConGolog programs. The proof is completely analogous to the proof of theorem 10.4.4, except for the case of prioritised parallel composition. In the latter case, use theorem 10.4.4. □

10.4.4 Basic Actions, Progression and Belief Bases

A third difference between ConGolog and 3APL is that the operational semantics of 3APL explicitly refers to states called *belief bases* whereas the axiomatic definition of the predicate *Trans* only mentions situations which *denote* such a state. In the operational semantics for 3APL, belief bases are updated by basic actions. The semantics of basic actions in ConGolog, however, is provided by successor state axioms in a given basic action theory.

For our purposes, we need a way to link successor state axioms to an update semantics for actions. This link is provided by the work of Lin and Reiter on the progression of databases (Lin & Reiter 1997). They define a progression operator for (relatively) complete basic action theories. This progression operator can be used in this context to specify the transition function $\mathcal{T}$ for 3APL basic actions.

Definition 10.4.6 *(progression operator)*
The progression operator *Prog* is defined by:

$$
\begin{array}{rcl}
\mathit{Prog}(\sigma, \epsilon) & = & \sigma, \\
\mathit{Prog}(\sigma, \mathsf{a}(\vec{t})) & = & \\
\multicolumn{3}{l}{\quad\{P(\vec{t}) \mid \sigma \models P(\vec{t}) \text{ and } P(\vec{t}) \text{ is a situation-independent sentence}\} \cup} \\
\multicolumn{3}{l}{\quad\{\neg P(\vec{t}) \mid \sigma \models \neg P(\vec{t}) \text{ and } P(\vec{t}) \text{ is a situation-independent sentence}\} \cup} \\
\multicolumn{3}{l}{\quad\{F(\vec{t'}, \mathit{now}) \mid \sigma \models \Phi_F(\vec{t'}, \mathsf{a}(\vec{t}), \mathit{now})\} \cup} \\
\multicolumn{3}{l}{\quad\{\neg F(\vec{t'}, \mathit{now}) \mid \sigma \models \neg\Phi_F(\vec{t'}, \mathsf{a}(\vec{t}), \mathit{now})\}}
\end{array}
$$

By theorem 3 in (Lin & Reiter 1997), the progression operator as defined in definition 10.4.6 yields a progression of a complete belief or data base σ since complete data bases are special cases of relatively complete databases and because we assume domain closure. A progression of a database by performing an action thus provides the update semantics of that action. By theorem 1 in (Lin & Reiter 1997), we then know that any sentence $\varphi[do(\alpha, S)]$ uniform in $do(\alpha, S)$ is implied by $\mathcal{A} + \sigma[S]$ iff $\mathcal{A} + \sigma[do(\alpha, S)]$ also implies $\varphi[do(\alpha, S)]$.

We can use the progression operator to specify a transition function for basic actions in 3APL. We define an update action in 3APL for every basic action A in the theory $\mathcal{A}$. The semantics of basic actions in 3APL is given by a semantic function $\mathcal{T}$ which defines in which states an action is enabled and what the resulting state of executing the action in that particular state is. $\mathcal{T}$ is defined as a partial function, which incorporates both the information of precondition and successor state axioms.

Definition 10.4.7 *(semantic function $\mathcal{T}$)*
Let $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory, and S be a closed term of sort *situation*. Define for every action $\mathsf{a}(\vec{t})$ its update semantics by:

$$
\begin{array}{ll}
\mathcal{T}(\mathsf{a}(\vec{t}), \sigma) = \mathit{Prog}(\sigma, \mathsf{a}(\vec{t})) & \text{if } \mathcal{A} + \sigma[S] \models \mathit{Poss}(\mathsf{a}(\vec{t}), S), \\
\mathcal{T}(\mathsf{a}(\vec{t}), \sigma) \text{ is undefined} & \text{otherwise.}
\end{array}
$$

Because instances of action precondition axioms must be uniform in a situation S, it follows by lemma 10.4.2 that $\mathcal{T}$ is well-defined. The main point of this definition is that it shows how to reduce situations (action histories) to *states* for complete databases.

10.4.5 Translating ConGolog programs into 3APL agents

Now we have set the stage, we can define a translation function τ from ConGolog programs to 3APL agents. The mapping τ defined below is defined by induction on the structure of programs. One of the more interesting cases is the translation of programs of the form $\pi x.\delta$ which are mapped onto a sequential 3APL program $random(x);\ \tau(\delta)$. The π operator is simulated by the special action *random* in 3APL, and the explicit binding by the π operator is replaced by the implicit binding mechanism in 3APL.

However, for this mapping to work we need to make sure that different occurrences of π operators are mapped onto different *random* actions even if they bind the same variable. That is, in a ConGolog program two occurrences of πx need to be mapped onto different *random* actions, say $random(x)$ and $random(y)$. The reason is that in 3APL an implicit binding mechanism is used and the first occurrence of a variable implicitly binds all later occurrences. If different occurrences of the same π operator are mapped onto the same *random* action, this might therefore result in bindings between variables that are not related in the ConGolog program. For this reason, *we assume that the translation function τ maps different occurrences of the π operator onto different random actions.*

Definition 10.4.8 *(translation function τ)*
The translation function τ is inductively defined by:

- $\tau(nil) = E$,
- $\tau(\mathsf{a}(\vec{t})) = \mathsf{a}(\vec{t})$,
- $\tau(\phi?) = \phi?$,
- $\tau(\delta_1;\ \delta_2) = \tau(\delta_1);\ \tau(\delta_2)$,
- $\tau(\delta_1 \mid \delta_2) = \tau(\delta_1) + \tau(\delta_2)$,
- $\tau(\pi x.\delta) = random(x);\ \tau(\delta)$,
- $\tau(\delta_1 \| \delta_2) = \tau(\delta_1) \| \tau(\delta_2)$,
- $\tau(\delta_1 \rangle\!\rangle \delta_2) = \tau(\delta_1) \rangle\!\rangle \tau(\delta_2)$,
- $\tau(P(\vec{t})) = p(\vec{t})$,
- $\tau(\mathbf{proc}\ P(\vec{x})\ \ \delta_P\ \mathbf{end}) = p(\vec{x}) \leftarrow \tau(\delta_P)$.

In the definition of τ, for simplicity, we did not incorporate the fact that τ needs to map different occurrences of πx operators onto different variables in a 3APL program (cf. the clause for $\pi x.\delta$). Technically, the assumption that τ maps different occurrences of a πx operator onto different *random* actions can be realised as follows. An infinite set V of variables can be associated with the τ function. The idea is that any occurrence of a π operator then is mapped by τ_V onto a $random(x)$ action such that $x \in V$. In case of a composed program, the set V needs to be partitioned in two (infinite) subsets. For example, $\tau_V(\delta_1;\ \delta_2) = \tau_{V_1}(\delta_1);\ \tau_{V_2}(\delta_2)$ such that $V = V_1 \cup V_2$, $V_1 \cap V_2 = \varnothing$, and both V_1 and V_2 are infinite. The details, however, are not very interesting and we do not provide them here. Throughout, we therefore assume that variable renaming is applied where necessary and appropriate. Recall that the framework for bisimulation introduced in chapter 8 explicitly allowed for steps that compensate for small syntactic differences. So-called α-steps were introduced for this purpose. Below, we will only mention that it is necessary to perform such steps at the most relevant places and assume implicitly that such steps are performed everywhere else when needed.

Finally, note that a ConGolog procedure is translated to a 3APL rule without a guard. Also notice that the translation function τ is defined on the set of open programs P, and not just on the set of closed ConGolog programs.

10.5 Embedding ConGolog in 3APL

The embedding of ConGolog in 3APL now proceeds in three stages. First, we show how to embed ConGolog programs without procedure calls or (prioritised) parallel composition in 3APL. In Section 6, we then extend this result to programs with procedure calls (this proof involves induction on the rank of a program). Finally, in section 7 we discuss the special problems associated with simulating parallel ConGolog programs in 3APL and show how to solve these problems. The main result, however, is established in this section where we prove an embedding result for all the basic constructs of ConGolog. We begin by showing that the concept defined by the *Final* predicate (for arbitrary ConGolog programs) coincides with termination of the corresponding 3APL τ-translation of the program modulo a number of silent steps.

Lemma 10.5.1 Let $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory, $\varphi \in \mathcal{L}_{now}$, and S be a closed situation term. Then, for any action theory $\mathcal{A}$ and ConGolog program δ we have that:

$$\mathcal{A} + \sigma[S] \models \mathit{Final}(\delta, S) \text{ iff } \langle \tau(\delta), \sigma \rangle \xrightarrow{i}^{*} \langle E, \sigma \rangle$$

Proof: Easy proof by induction on the rank and structure of a program. Use domain closure for the case that δ is a nondeterministic choice of argument program. $\square$

The main embedding result is the following theorem, which shows that the transition relation defined by the *Trans* predicate bisimulates with the step relation $\Longrightarrow$.

Theorem 10.5.2 *(bisimulation theorem)*
Let δ and δ' be (closed) ConGolog programs, $\sigma \subseteq \mathcal{L}_{now}$ be a complete theory, and S be a closed term of sort *situation*. Then:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta, S, \delta', do(\alpha, S))$$
$$\text{iff}$$
$$\langle \tau(\delta), \sigma \rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$$

Proof: We prove the theorem by induction on the structure of programs.

Basic Actions: $\delta = \mathsf{a}(\vec{t})$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\mathsf{a}(\vec{t}), S, \delta', \mathit{do}(\mathsf{a}(\vec{t}), S))$$
$$\text{iff}$$
$$\langle \mathsf{a}(\vec{t}), \sigma\rangle \overset{\mathsf{a}(\vec{t})}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \mathsf{a}(\vec{t}))\rangle$$

Proof:

($\Rightarrow$) By definition of the *Trans* predicate, we have $\mathcal{A} + \sigma[S] \models \mathit{Poss}(\mathsf{a}(\vec{t}), S) \wedge \delta' = \mathit{nil}$. By definition of $\mathcal{T}$ and τ, we then have $\mathcal{T}(\mathsf{a}(\vec{t}), \sigma) = \mathit{Prog}(\sigma, \mathsf{a}(\vec{t}))$ and $\tau(\mathit{nil}) = E$. From this we obtain $\langle \mathsf{a}(\vec{t}), \sigma\rangle \overset{\mathsf{a}(\vec{t})}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \mathsf{a}(\vec{t}))\rangle$.

($\Leftarrow$) By the transition rule for basic actions, we must have that $\mathcal{T}(\mathsf{a}(\vec{t}), \sigma) = \mathit{Prog}(\sigma, \mathsf{a}(\vec{t}))$. This implies that $\mathcal{A} + \sigma[S] \models \mathit{Poss}(\mathsf{a}(\vec{t}), S)$. It follows from the definition of *Trans* that $\mathcal{A} + \sigma[S] \models \mathit{Trans}(\mathsf{a}(\vec{t}), S, \mathit{nil}, \mathit{do}(\mathsf{a}(\vec{t}), S))$.

Tests: $\delta = \phi?$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\phi?, S, \delta', \mathit{do}(\epsilon, S)) \text{ iff } \quad \langle \phi?, \sigma\rangle \overset{\epsilon}{\Longrightarrow} \langle \tau(\delta'), \sigma\rangle$$

Proof:

($\Rightarrow$) By definition of *Trans*, we have $\mathcal{A} + \sigma[S] \models \varphi[S] \wedge \delta' = \mathit{nil}$. Since $\varphi[S]$ is closed and uniform in S, and $\tau(\mathit{nil}) = E$, by lemma 10.4.2 we obtain $\sigma \models \varphi\varnothing$, which is the required premise of the transition rule for tests.

($\Leftarrow$) By the transition rule for tests, we must have $\sigma \models \varphi$ since φ is closed, and so we also have that $\mathcal{A} + \sigma[S] \models \varphi[S]$ by lemma 10.4.2. It follows from the definition of *Trans* that $\mathcal{A} + \sigma[S] \models \mathit{Trans}(\phi?, S, \mathit{nil}, \mathit{do}(\epsilon, S))$.

Sequential Composition: $\delta = \delta_1;\ \delta_2$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1;\ \delta_2, S, \delta', \mathit{do}(\alpha, S))$$
$$\text{iff}$$
$$\langle \tau(\delta_1;\ \delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$$

Proof: By the induction hypothesis, we may assume that we know that:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1, S, \delta_1', \mathit{do}(\alpha, S)) \text{ iff } \langle \tau(\delta_1), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha)\rangle$$

and

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_2, S, \delta_2', \mathit{do}(\alpha, S)) \text{ iff } \langle \tau(\delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta_2'), \mathit{Prog}(\sigma, \alpha)\rangle$$

($\Rightarrow$) We prove the implication by reasoning by cases: (which is allowed by theorem 10.4.5)

1. First, suppose $\mathcal{A}+\sigma[S] \models \exists\,\gamma.\delta' = (\gamma;\ \delta_2) \wedge \mathit{Trans}(\delta_1, S, \gamma, \mathit{do}(\alpha, S))$. This implies there is a ConGolog program δ_1' such that $\mathcal{A}+\sigma[S] \models \delta' = (\delta_1';\ \delta_2) \wedge \mathit{Trans}(\delta_1, S, \delta_1', \mathit{do}(\alpha, S))$. By the induction hypothesis, we then obtain: $\langle \tau(\delta_1), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha)\rangle$. From this and the transition rule for sequential composition, we conclude that $\langle \tau(\delta_1;\ \delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta_1';\ \delta_2), \mathit{Prog}(\sigma, \alpha)\rangle$. Since $\delta' = (\delta_1';\ \delta_2)$, we obtain: $\langle \tau(\delta_1;\ \delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$.
2. Secondly, suppose that we have $\mathcal{A}+\sigma[S] \models \neg\exists\,\gamma.\delta' = (\gamma;\ \delta_2) \wedge \mathit{Trans}(\delta_1, S, \gamma, \mathit{do}(\alpha, S))$. By the axiomatic definition of *Trans* for ; , we then know that $\mathcal{A}+\sigma[S] \models \mathit{Final}(\delta_1, s) \wedge \mathit{Trans}(\delta_2, S, \delta', \mathit{do}(\alpha, S))$. Since $\mathcal{A}+\sigma[S] \models \mathit{Final}(\delta_1, S)$, by lemma 10.5.1, we conclude that $\langle \tau(\delta_1), \sigma\rangle \xrightarrow{i}{}^{*} \langle E, \sigma\rangle$. By means of the induction hypothesis, we then obtain that $\langle \tau(\delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$. By the definition of $\Longrightarrow$ we then may add the silent steps of δ_1 in front of the computation involving δ_2 and we obtain: $\langle \tau(\delta_1;\ \delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$.

($\Leftarrow$) From $\langle \tau(\delta_1;\ \delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$ it follows that: (i) $\langle \tau(\delta_1), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha)\rangle$ for some δ_1', or (ii) $\langle \tau(\delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$ and $\langle \tau(\delta_1), \sigma\rangle \xrightarrow{i}{}^{*} \langle E, \sigma\rangle$. In the first case, simply apply the induction hypothesis and note that δ' must be of the form $\delta_1';\ \delta_2$ to conclude that $\mathcal{A}+\sigma[S] \models \mathit{Trans}(\delta_1;\ \delta_2, S, \delta', \mathit{do}(\alpha, S))$. In the second case, use lemma 10.5.1 to obtain $\mathcal{A}+\sigma[S] \models \mathit{Final}(\delta_1, S)$ and apply the induction hypothesis.

Nondeterministic Choice: $\delta = \delta_1 + \delta_2$:

$$\mathcal{A}+\sigma[S] \models \mathit{Trans}(\delta_1 \mid \delta_2, S, \delta', \mathit{do}(\alpha, S))$$
$$\text{iff}$$
$$\langle \tau(\delta_1) + \tau(\delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \sigma'\rangle$$

Proof:

$$\mathcal{A}+\sigma[S] \models \mathit{Trans}(\delta_1 \mid \delta_2, S, \delta', \mathit{do}(\alpha, S))$$

By the axiomatic definition of *Trans* this is equivalent to:

$$\mathcal{A}+\sigma[S] \models \mathit{Trans}(\delta_1, S, \delta', \mathit{do}(\alpha, S)) \vee \mathit{Trans}(\delta_2, S, \delta', \mathit{do}(\alpha, S))$$

Now, by theorem 10.4.5 and the fact that σ is complete, the latter is equivalent to:

$\mathcal{A}+\sigma[S] \models \mathit{Trans}(\delta_1, S, \delta', \mathit{do}(\alpha, S))$ or
$\mathcal{A}+\sigma[S] \models \mathit{Trans}(\delta_2, S, \delta', \mathit{do}(\alpha, S))$

This in turn, by the induction hypothesis, is equivalent to:

$$\langle \tau(\delta_1), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle \text{ or } \langle \tau(\delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$$

Finally, this is equivalent to:

$$\langle \tau(\delta_1) + \tau(\delta_2), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha)\rangle$$

Nondeterministic Choice of Argument: $\delta = \pi x.\delta'$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}((\pi x.\delta'), S, \delta'', \mathit{do}(\alpha, S)) \quad \text{iff} \quad \langle \tau(\pi x.\delta'), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta''), \sigma'\rangle$$

Proof:

(⇒)

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}((\pi x.\delta'), S, \delta'', \mathit{do}(\alpha, S))$$

By the axiomatic definition of *Trans* this is equivalent to:

$$\mathcal{A} + \sigma[S] \models \exists\, x.\mathit{Trans}(\delta', S, \delta'', \mathit{do}(\alpha, S)))$$

Because of domain closure, there is a ground witness t for $\exists\, x$ and we obtain:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta'\{x = t\}, S, \delta'', \mathit{do}(\alpha, S))$$

By the induction hypothesis, we then have that

$$\langle \tau(\delta'\{x = t\}), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta''), \mathit{Prog}(\alpha, \sigma)\rangle$$

From this, and because $\mathit{random}(x)$ gives rise to a silent step, we can then derive:

$$\langle \mathit{random}(x);\ \tau(\delta'), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta''), \sigma'\rangle$$

By definition of the translation function τ, we obtain:

$$\langle \tau(\pi x.\delta'), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta''), \sigma'\rangle$$

(⇐) From

$$\langle \mathit{random}(x);\ \tau(\delta'), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta''), \mathit{Prog}(\sigma, \alpha)\rangle$$

it follows that:

$$\langle \tau(\delta\{x = t\}), \sigma\rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta''), \mathit{Prog}(\sigma, \alpha)\rangle$$

for some t, since $\mathit{random}(x)$ gives rise to an silent step. By the induction hypothesis, we then obtain:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta'\{x = t\}, S, \delta'', \mathit{do}(\alpha, S))$$

By the semantic definition of $\exists\, x$ we then can derive:

$$\mathcal{A} + \sigma[S] \models \exists\, x.\, \mathit{Trans}(\delta', S, \delta'', \mathit{do}(\alpha, S))$$

And finally, by the axiomatic definition of *Trans* this is equivalent to:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}((\pi x.\delta'), S, \delta'', \mathit{do}(\alpha, S))$$

As a corollary, we obtain that ConGolog programs (without procedures or parallelism) and the translations of these programs into 3APL compute the same belief or data bases and compute these belief bases by executing the same sequence of actions:

Corollary 10.5.3 Let $\mathcal{A}$ be a basic action theory with initial database $\sigma[S_0]$, and α be a sequence of basic actions.

$$\mathcal{A} + \sigma[S_0] \models \mathit{Trans}^*(\delta, S_0, \mathit{nil}, \mathit{do}(\alpha, S_0))$$
$$\text{iff}$$
$$\langle \tau(\delta), \sigma \rangle \overset{\alpha}{\Longrightarrow}{}^* \langle E, \mathit{Prog}(\sigma, \alpha) \rangle$$

where *Trans** denotes the transitive closure of *Trans*.

10.6 Procedures

In this section, we extend the bisimulation result to include programs with procedures. The proof proceeds by induction on the rank and structure of a program. The proofs for programs without procedure calls are essentially the same as in the previous section. The only interesting new case is that of simulating a procedure call.

Procedure Call: $\delta = P(\vec{t})$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(P(\vec{t}), S, \delta', \mathit{do}(\alpha, S)) \quad \text{iff} \quad \langle \tau(P(\vec{t})), \sigma \rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \sigma' \rangle$$

Proof: We proceed by induction on the rank of a program. The base case, programs with rank 0, coincides with programs without procedure calls, and is proven in the previous section. Assuming that we know the simulation result holds for all programs of rank n, we now show that it also holds for programs with rank $n+1$:

($\Rightarrow$) Suppose $P(\vec{t})$ is of rank $n+1$ and $\mathcal{A}+\sigma[S] \models \mathit{Trans}(P(\vec{t}), S, \delta', \mathit{do}(\alpha, S))$. Then we also have $\mathcal{A}+\sigma[S] \models \mathit{Trans}(\delta_P{}^{\vec{v}}_{\vec{t}}, S, \delta', \mathit{do}(\alpha, S))$ where $\delta_P{}^{\vec{v}}_{\vec{t}}$ is the body of procedure P with formal parameters replaced with actual parameters. Moreover, if P is of rank $n+1$, then δ_P is of rank n. Then, by the induction hypothesis, we obtain: $\langle \tau(\delta_P{}^{\vec{v}}_{\vec{t}}), \sigma \rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$. By the transition rule for rule application, we then derive: $\langle \tau(P(\vec{t})), \sigma \rangle \overset{\alpha}{\Longrightarrow}$

$\langle\tau(\delta'), \mathit{Prog}(\sigma,\alpha)\rangle$, because application of a rule is an silent step. Notice that in particular in a rule application step of 3APL, variables may have been renamed. This may result in a mismatch between the variables introduced by the renaming mechanism of 3APL and the translation function τ. However, to compensate for such small differences, we may perform so-called α-steps that rename variables.

($\Leftarrow$) Suppose $\langle\tau(P(\vec{t})),\sigma\rangle \overset{\alpha}{\Longrightarrow} \langle\tau(\delta'), \mathit{Prog}(\sigma,\alpha)\rangle$. Then there must be some rule such that: $\langle\tau(P(\vec{t})),\sigma\rangle \longrightarrow \langle\tau(\delta_P{}^{\vec{v}}_{\vec{t}}),\sigma\rangle$. Because $P(\vec{t})$ is of rank $n+1$, $\delta_P{}^{\vec{v}}_{\vec{t}}$ must be of rank n. By the induction hypothesis, we then have: $\mathcal{A}+\sigma[S] \models \mathit{Trans}(\delta_P{}^{\vec{v}}_{\vec{t}}, S, \delta', do(\alpha,S))$. From this we immediately obtain: $\mathcal{A}+\sigma[S] \models \mathit{Trans}(P(\vec{t}), S, \delta', do(\alpha,S))$.

$\square$

10.7 Parallel Composition

In this section, we extend the bisimulation result to *arbitrary ConGolog programs* including parallel as well as prioritised parallel programs. To simulate prioritised parallel programs 3APL is extended with the $\rangle\rangle$ operator and we show how to define the semantics of this operator in a transition style semantics below. The parallel composition of goals is included in 3APL. The semantics of the parallel operator $||$ in 3APL, however, differs from that of the ConGolog semantics. Because of this difference, the main issue in extending the simulation result to parallel programs is to prove that the two semantics are equivalent with respect to some appropriate observation criterion. That is, we must show that parallel programs in 3APL and ConGolog compute the same things.

The difference in the semantic definitions of the parallel operator concerns the ordering of computation steps of a 3APL (parallel) program. A 3APL program can perform silent steps which do not have a counterpart in the ConGolog semantics. The problem concerns the order in which these silent steps are performed. This can be illustrated as follows: given a parallel program $\delta_1||\delta_2$, in the ConGolog semantics only δ_1 *or* δ_2 can be transformed in a single step but *not* both, while according to the 3APL semantics as defined by $\Longrightarrow$ both subprograms may perform silent steps *before* an action or a test (a 'real' ConGolog step) is performed. For example, the ConGolog program

```
proc P()
    a
end
proc Q()
    b
end
P()||Q()
```

can execute either the left or right branch of the parallel composition resulting in respectively $Q()$ and $P()$ as the remaining programs for execution. The program is translated to $p()\|q()$ in 3APL (plus translations of the procedure definitions to rules). According to the $\Longrightarrow$ semantics which abstracts from silent steps, this program, however, can result in either $p()$, $q()$, a or b after performing one $\Longrightarrow$ step. The latter two new possibilities result from the fact that with respect to the $\Longrightarrow$ semantics a silent step may have been performed in which the procedure is expanded into its body before an actual step (not a silent step) is performed. The $\Longrightarrow$ transition semantics thus allows silent steps of *both* subprograms to be performed before an actual step is performed in either one of them, whereas we would like to make sure that only computation steps associated with one of the subprograms are performed.

To solve this problem we show that the order of performing silent steps does not matter. For this purpose, we introduce a new transition relation $\rightsquigarrow$ which imposes a restriction on the order in which silent steps are performed in a parallel program and show that $\rightsquigarrow$ and $\longrightarrow$ are equivalent in the sense that they compute the same belief bases (our observation criterion).

The transition relation $\rightsquigarrow$ is derived from $\Longrightarrow$. In the definition of the $\rightsquigarrow$ transition relation also a specification of the semantics of the prioritised parallel operator is given. The transition rule for prioritised parallel composition $\pi_1 \rangle\!\rangle \pi_2$ uses a negative premise to specify that the execution of π_2 is only allowed if π_1 cannot perform an action or test after a finite number of silent steps. A justification for this type of transition rule can be found in (Groote 1993).

Definition 10.7.1 *(transition relation $\rightsquigarrow$)*
Let *Parfree* denote the set of 3APL programs without occurrences of parallel operators.

$$\frac{\langle \pi, \sigma\rangle \overset{l}{\Longrightarrow}_\gamma \langle \pi', \sigma'\rangle, \pi \in \mathit{Parfree}}{\langle \pi, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi', \sigma'\rangle}$$

$$\frac{\langle \pi_1, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_1', \sigma'\rangle}{\langle \pi_1;\ \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_1';\ \pi_2\gamma, \sigma'\rangle}$$

$$\frac{\langle \pi_1, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_1', \sigma'\rangle}{\langle \pi_1 + \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_1', \sigma'\rangle} \qquad \frac{\langle \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_2', \sigma'\rangle}{\langle \pi_1 + \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_2', \sigma'\rangle}$$

$$\frac{\langle \pi_1, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_1', \sigma'\rangle}{\langle \pi_1 \| \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_1' \| \pi_2\gamma, \sigma'\rangle} \qquad \frac{\langle \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_2', \sigma'\rangle}{\langle \pi_1 \| \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_1\gamma \| \pi_2', \sigma'\rangle}$$

$$\frac{\langle \pi_1, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_1', \sigma'\rangle}{\langle \pi_1 \rangle\!\rangle \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_1' \rangle\!\rangle \pi_2\gamma, \sigma'\rangle}$$

$$\frac{\forall\, \pi_1', \sigma', \theta, m(\langle \pi_1, \sigma\rangle \overset{m}{\not\rightsquigarrow}_\theta \langle \pi_1', \sigma'\rangle) \text{ and } \langle \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow}_\gamma \langle \pi_2', \sigma''\rangle}{\langle \pi_1 \rangle\!\rangle \pi_2, \sigma\rangle \overset{l}{\rightsquigarrow} \langle \pi_1 \gamma \rangle\!\rangle \pi_2', \sigma''\rangle}$$

The following theorem provides the basis for our simulation result. It proves that the transition relation $\rightsquigarrow$ is equivalent to $\Longrightarrow$ with respect to computed belief bases, which is used as the observation criterion here for programs without prioritised parallel composition (which is not defined for $\Longrightarrow$).

Theorem 10.7.2 Let π be the translation of a ConGolog program δ without prioritised parallel composition, i.e. $\pi = \tau(\delta)$. Then:

$\langle \pi, \sigma\rangle \Longrightarrow^* \langle E, \sigma'\rangle$ iff $\langle \pi, \sigma\rangle \rightsquigarrow^* \langle E, \sigma'\rangle$

Proof:

($\Rightarrow$) By induction on the length of the computation and the structure of π. The base case, a computation of length 1, is trivial, since in that case π must be $\mathsf{a}(\vec{t})$ or $\phi?$. So, suppose for all computations of length n the theorem holds. We must prove it for computations of length $n+1$.

- Basic Actions, Tests: Easy.
- Random Action: Is not a translation of a ConGolog program.
- Sequential Composition: Easy; partition computation into $\overset{\alpha}{\Longrightarrow}$ steps.
- Nondeterministic Choice: Easy.
- Parallel Composition: $\pi = \pi_1 \| \pi_2$.
  Take the first m steps of the computation such that the mth step is a $\mathsf{a}(\vec{t})$ or $\phi?$ step and all previous steps are silent steps. The mth step is either performed by π_1 or by π_2. Suppose it is performed by π_1 (the other case is analogous). By rearranging the first m steps such that all steps performed by π_1 are performed first - in the same order - and then performing the (silent) steps performed by π_2, we still have a legal computation which does not change the computed result (since the steps from π_2 only expand procedure calls or randomly guess values). Now, the sequence of π_1 steps correspond to one $\Longrightarrow$ step by definition. The remaining computation is at least one computation step shorter, and thus we are done.
- Procedure Call: Use induction hypothesis.

($\Leftarrow$) Trivial.

□

Now we are able to extend the simulation result to (prioritised) parallel programs. To prove this extended simulation result, we use the new transition relation $\rightsquigarrow$. The proof for all cases except for parallel and prioritised parallel composition are analogous to the proofs of previous sections and are omitted for this reason. Theorem 10.7.3 shows that the transition relation defined by *Trans* for arbitrary ConGolog programs bisimulates with the step relation $\rightsquigarrow$.

Theorem 10.7.3 *(bisimulating parallel and prioritised parallel composition)*

Parallel Composition: $\delta = \delta_1 \| \delta_2$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1 \| \delta_2, S, \delta', do(\alpha, S))$$
$$\text{iff}$$
$$\langle \tau(\delta_1) \| \tau(\delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$$

Proof: Note that the base case, i.e. $\langle \pi, \sigma \rangle \rightsquigarrow \langle \pi', \sigma' \rangle$ because $\langle \pi, \sigma \rangle \Longrightarrow \langle \pi', \sigma' \rangle$, has been proven in previous sections. We now deal with the remaining cases.

($\Rightarrow$) Assume $\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1 \| \delta_2, S, \delta', do(\alpha, S))$. We need to distinguish two cases, the case where δ_1 is executed and the case where δ_2 is executed. Because of symmetry, we only give the details for one of these cases. So, assume: $\mathcal{A} + \sigma[S] \models \exists \gamma . \delta' = (\gamma \| \delta_2) \wedge \mathit{Trans}(\delta_1, S, \gamma, do(\alpha, S))$. As a consequence, there is a δ_1' such that: $\mathcal{A} + \sigma[S] \models \delta' = (\delta_1' \| \delta_2) \wedge \mathit{Trans}(\delta_1, S, \delta_1', do(\alpha, S))$. Then, by the induction hypothesis, we obtain: $\langle \tau(\delta_1), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha) \rangle$. By definition, we then have: $\langle \tau(\delta_1) \| \tau(\delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$.

($\Leftarrow$) Assume $\langle \tau(\delta_1) \| \tau(\delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$. In that case, by definition 10.7.1 we know that either $\langle \tau(\delta_1), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha) \rangle$ for some δ_1' such that $\delta' = \delta_1' \| \delta_2$, or $\langle \tau(\delta_2), \sigma \rangle \overset{\alpha}{\Longrightarrow} \langle \tau(\delta_2'), \mathit{Prog}(\sigma, \alpha) \rangle$ for some δ_2' such that $\delta' = \delta_1 \| \delta_2'$. Then apply the induction hypothesis.

Prioritised Parallel Composition: $\delta = \delta_1 \rangle\!\rangle \delta_2$:

$$\mathcal{A} + \sigma[S] \models \mathit{Trans}(\delta_1 \rangle\!\rangle \delta_2, S, \delta', do(\alpha, S))$$
$$\text{iff}$$
$$\langle \tau(\delta_1 \rangle\!\rangle \delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$$

Proof:

($\Rightarrow$) We prove the implication by reasoning by cases: (which is allowed by theorem 10.4.5)

1. First, suppose $\mathcal{A} + \sigma[S] \models \exists \gamma . \delta' = (\gamma \rangle\!\rangle \delta_2) \wedge \mathit{Trans}(\delta_1, S, \gamma, do(\alpha, S))$. Suppose δ_1' is a ConGolog program which satisfies this equation. It then follows from the induction hypothesis that we have $\langle \tau(\delta_1), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_1'), \mathit{Prog}(\sigma, \alpha) \rangle$. By the transition rule for prioritised parallel composition of definition 10.7.1, this implies that $\langle \tau(\delta_1 \rangle\!\rangle \delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), \mathit{Prog}(\sigma, \alpha) \rangle$.

2. Secondly, assume we have that $\mathcal{A} + \sigma[S] \models \neg \exists \gamma . \delta' = (\gamma \rangle\rangle \delta_2) \wedge Trans(\delta_1, S, \gamma, do(\alpha, S))$. By the axiomatic definition of *Trans*, we then have that $\mathcal{A}+\sigma[S] \models \exists \gamma . \delta' = (\delta_1 \rangle\rangle \gamma) \wedge Trans(\delta_2, S, \gamma, do(\alpha, S)) \wedge \neg \exists \eta, s''. Trans(\delta_1, S, \eta, s'')$. By the induction hypothesis, we then have that $\langle \tau(\delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_2'), Prog(\sigma, \alpha) \rangle$ for some δ_2' such that $\delta' = \delta_1 \rangle\rangle \delta_2'$. Moreover, there is no transition $\rightsquigarrow$ corresponding to δ_1. By the transition rule for prioritised parallel composition, we then have that $\langle \tau(\delta_1 \rangle\rangle \delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), Prog(\sigma, \alpha) \rangle$.

($\Leftarrow$) From $\langle \tau(\delta_1 \rangle\rangle \delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta'), Prog(\sigma, \alpha) \rangle$ it follows that: (i) $\langle \tau(\delta_1), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_1'), Prog(\sigma, \alpha) \rangle$ for some δ_1' or (ii) $\langle \tau(\delta_2), \sigma \rangle \overset{\alpha}{\rightsquigarrow} \langle \tau(\delta_2'), Prog(\sigma, \alpha) \rangle$ for some δ_2' and there is no transition associated with δ_1. In both cases, use the induction hypothesis to conclude $\mathcal{A}+\sigma[S] \models Trans(\delta_1 \rangle\rangle \delta_2, S, \delta', do(\alpha, S))$.

□

Finally, we obtain that *arbitrary* ConGolog programs and the translations of these programs in 3APL compute the same belief or data bases and compute these belief bases by executing the same sequence of actions. The extended version of corollary 10.5.3 now includes all ConGolog programs.

Corollary 10.7.4 Let $\mathcal{A}$ be a basic action theory with initial database $\sigma[S_0]$, and α be a sequence of basic actions.

$$\mathcal{A} + \sigma[S_0] \models Trans^*(\delta, S_0, nil, do(\alpha, S_0))$$
$$\text{iff}$$
$$\langle \tau(\delta), \sigma \rangle \overset{\alpha}{\rightsquigarrow}{}^{*} \langle E, Prog(\sigma, \alpha) \rangle$$

□

10.8 Discussion

ConGolog is a programming language which extends basic action theories in the situation calculus with operators for building composed programs. The logical perspective of the situation calculus offers a very expressive framework for specifying agents. Basic action theories provide a framework for specifying actions and offer a solution to the frame problem. The logical semantics of ConGolog, however, does not straightforwardly provide an implementation language, in contrast with the operational semantics of 3APL. The embedding result of this chapter shows that one option to implement (a restricted version of) ConGolog is to embed the language into 3APL. Another important feature of the logical semantics is that in the presence of functional fluents, situations cannot be identified with states.

In contrast, with respect to 3APL a clear distinction is made between the programming language and a programming logic (cf. Hindriks et al. (n.d.)) for proving properties of 3APL agents. The agent language 3APL abstracts both from the knowledge representation that agents use and a concrete specification of actions. The embedding result shows that basic action theories in the situation calculus can be used to specify actions and to derive an update semantics for 3APL actions.

Both languages emphasise different aspects of agent computing. ConGolog is presented as a high-level programming alternative to planning. The focus is on extracting a legal action sequence from a nondeterministic program. A ConGolog program thus is seen as a vehicle for computing a situation (action history). As in planning, finding a legal action sequence requires search and this explains the use of a backtracking model of execution. The backtracking model is inherited from logic programming, which is used to implement ConGolog (Giacomo et al. 2000).

With respect to 3APL, the focus is on computing belief bases. Upon termination a 3APL program returns a belief base. The execution model that is proposed is that of the 'imperative flow of control'. The basic feature of this model is that a commitment to a choice is made as soon as an action has been executed. Because of the embedding result, it is clear, however, that neither the semantics of ConGolog nor that of 3APL dictates the use of one or the other model of execution (cf. also Giacomo & Levesque (1998)).

Finally, the construction of an embedding of ConGolog into 3APL also identified several common and distinguishing features of the formalisms used to specify a semantics. Basically, the embedding result indicates that an axiomatic definition in an extended predicate logic like the situation calculus and a Plotkin-style transition semantics result in more or less equivalent semantics. The use of a logical semantics specified in the situation calculus, however, requires careful consideration of a number of issues, like the incompleteness of databases and domain closure. Moreover, the logical semantics seems suitable for proving partial correctness, but raises some doubts as to its usefulness for proving termination properties.

Part III:

Agent Programming with Declarative Goals

In the first two parts, the metaphor of intelligent agents has been made precise by providing a programming language with a formal semantics and by comparing this language with other proposals for agent languages in the literature. A different approach to clarify the concept of an intelligent agent is to design *agent logics*. In the literature, a range of proposals of - mainly - modal logics for the specification of agents have been made. These logics typically define the core notions associated with intelligent agents like *belief*, *goal*, *intention*, etc. (Rao 1996*b*, Linder et al. 1996, Cohen & Levesque 1990*a*, Cohen & Levesque 1995).

The relation of these logics with the programming languages that we discussed so far has not been studied yet. As was shown in part II, the programming languages AGENT0, AgentSpeak(L), ConGolog and 3APL are closely related agent programming frameworks. The terminology may differ from case to case, but each of these programming languages define agents in terms of beliefs, goals, plans and capabilities. These agent languages thus are based on similar notions as those defined by the logical approaches. However, there is one notable difference. In agent logics, a goal is a *declarative* concept, whereas in the programming languages that we discussed goals are defined as sequences of actions or *plans*. Whether they are called commitments (AGENT0), intentions (AgentSpeak(L)), or goals (3APL) makes little difference: Each of these notions are structures built from *actions* and therefore similar in nature to *plans*. With respect to ConGolog, a more traditional computer science perspective is adopted, and the corresponding structures are simply called programs. The type of goal included in these languages may also be called a *goal-to-do* and provides for a *procedural* perspective on goals.

In part III, we aim to bridge this gap between agent logics and agent programming languages. It is our aim to incorporate *declarative* goals in an agent language. To this end, we introduce a new agent programming language called GOAL. GOAL provides a concrete proposal to bridge the gap between theory and practice. The GOAL framework offers a complete theory of agent programming in the sense that it provides both for a programming framework and a programming logic for intelligent agents.

CHAPTER 11

The Agent Language GOAL

A long and lasting problem in agent research has been to close the gap between agent logics and agent programming frameworks. The main reason for this problem of establishing a link between agent logics and agent programming frameworks, we believe, is that agent programming frameworks have not incorporated the concept of a *declarative goal*. Instead, programming frameworks have mainly focused on plans or *goals-to-do* instead of the end goals to be realised. These declarative goals are also called *goals-to-be*. In this chapter, a new programming language that is called GOAL (for Goal-Oriented Agent Language) is introduced. The motivational component of GOAL agents consists of declarative goals.

The fact that GOAL includes declarative goals distinguishes this language from other agent languages like AGENT0, AgentSpeak(L), ConGolog and 3APL. In this respect, it is also different from the PLACA language (Thomas 1993), a successor of AGENT0. PLACA also focuses more on extending AGENT0 to a language with complex planning structures than on providing a clear theory of declarative goals of agents as part of a programming language and in this respect is similar to AgentSpeak(L) and 3APL. The value of adding declarative goals to agent programming lies both in the fact that it offers a new abstraction mechanism as well as that agent programs with declarative goals more closely approximate the intuitive concept of an intelligent agent. To fully realise the potential of the notion of an intelligent agent, a declarative notion of a goal, therefore, should also be incorporated into agent programming languages.

First, the basic ideas behind the programming language GOAL are introduced. To provide GOAL with an operational semantics, a number of issues need to be solved. In particular, the interaction between the goal and belief bases of a GOAL agent must be clarified in a satisfactory way. For this purpose, the notion of a *commitment strategy* - one of the main theoretical insights gained from research on agent logics - is used. A commitment strategy explains the relation between beliefs and goals and can be used to construct a computational semantics for GOAL. Finally, an example GOAL agent is presented.

11.1 The Programming Language GOAL

In this section, we introduce the programming language GOAL. The programming language GOAL is inspired by work in concurrent programming, in particular by the language UNITY designed by Chandy and Misra (Chandy & Misra 1988). The basic idea is that a set of actions which execute in parallel constitutes a program. However, whereas UNITY is a language based on assignment to variables, the language GOAL is an agent-oriented programming language that incorporates more complex notions such as belief, goal, and agent capabilities which operate on high-level information instead of simple values.

As in most agent programming languages, GOAL agents select actions on the basis of their current mental state. A mental state consists of the beliefs and goals of the agent. However, in contrast to most agent languages, GOAL incorporates a *declarative* notion of a goal that is used by the agent to decide what to do. Both the beliefs and the goals are drawn from one and the same logical language, $\mathcal{L}$, with associated consequence relation $\models$. An agent thus keeps two databases, respectively called the *belief base* and the *goal base*. The difference between these two databases originates from the different meaning assigned to sentences stored in the belief base and sentences stored in the goal base. To clarify the interaction between beliefs and goals, one of the more important problems that needs to be solved is establishing a meaningful relationship between beliefs and goals. This problem is solved here by imposing a constraint on mental states that is derived from the default commitment strategy that agents use. The notion of a commitment strategy is explained in more detail below. The constraint imposed on mental states requires that an agent does not believe that ϕ is the case if it has a goal to achieve ϕ, and, moreover, requires ϕ to be consistent if ϕ is a goal.

Definition 11.1.1 *(mental state)*
A *mental state* of an agent is a pair $\langle\sigma,\gamma\rangle$ where $\sigma \subseteq \mathcal{L}$ are the agent's beliefs and $\gamma \subseteq \mathcal{L}$ are the agent's goals and σ and γ are such that for any $\phi \in \gamma$ we have:

- ϕ is not entailed by the agent's beliefs ($\sigma \not\models \phi$),
- ϕ is consistent ($\not\models \neg\phi$), and
- σ is consistent ($\sigma \not\models \mathsf{false}$).

A mental state does *not* contain a program or plan component in the 'classical' sense. Although both the beliefs and the goals of an agent are drawn from the same logical language, as we will see below, the formal meaning of beliefs and goals is very different. This difference in meaning reflects the different features of the beliefs and the goals of an agent. The declarative goals here are best thought of as *achievement* goals. That is, these goals *describe* a goal state that the agent desires to reach. Mainly due to the temporal features of such goals many properties of beliefs fail for goals. For example, the fact that an agent has the goal to *be at home* and the goal to *be at the movies* does not allow the conclusion that this agent also has the conjunctive goal to *be at home and at the movies* at the same time. As a consequence, less stringent consistency requirements are imposed on goals than on beliefs. An agent may have the goal to be at home and the goal to be at the movies simultaneously; assuming these two goals cannot consistently be achieved at the same time does not mean that an agent cannot have adopted both in the language GOAL.

With respect to GOAL, we assume that the language $\mathcal{L}$ used for representing beliefs and goals is a simple *propositional language*. As a consequence, we do not discuss the use of variables nor parameter mechanisms. Our motivation for this assumption is the fact that we want to present our main ideas in their simplest form and do not want to clutter the definitions below with too many details.

The language $\mathcal{L}$ for representing beliefs and goals is extended to a new language $\mathcal{L}_M$ which enables us to formulate conditions on the mental state of an agent. The language $\mathcal{L}_M$ consists of so called *mental state formulas*. A mental state formula is a boolean combination of the basic mental state formulas $\mathsf{B}\phi$, which expresses that ϕ is believed to be the case, and $\mathsf{G}\phi$, which expresses that ϕ is a goal of the agent.

Definition 11.1.2 *(mental state formula)*
The set of *mental state formulas* $\mathcal{L}_M$ is defined by:

- if $\phi \in \mathcal{L}$, then $\mathsf{B}\phi \in \mathcal{L}_M$,
- if $\phi \in \mathcal{L}$, then $\mathsf{G}\phi \in \mathcal{L}_M$,
- if $\varphi_1, \varphi_2 \in \mathcal{L}_M$, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2 \in \mathcal{L}_M$.

The usual abbreviations for the propositional operators $\vee$, $\rightarrow$, and $\leftrightarrow$ are used. We write true as an abbreviation for $\mathsf{B}(p \vee \neg p)$ for some p and false for $\neg\mathsf{true}$.

A third basic concept in GOAL is that of an agent *capability*. The capabilities of an agent consist of a set of so called *basic actions*. The effects of executing such a basic action are reflected in the beliefs of the agent and therefore a basic action is taken to be a belief update on the agent's beliefs. A basic action thus is a *mental state transformer*. Two examples of agent capabilities are the actions $\mathsf{ins}(\phi)$ for inserting ϕ in the belief base and $\mathsf{del}(\phi)$ for removing ϕ from the belief base. Agent capabilities are not supposed to change the goals of an agent, but because of the constraints on mental states they may as a side effect modify the current goals. For the purpose of modifying the goals of the agent, two special actions $\mathsf{adopt}(\phi)$ and $\mathsf{drop}(\phi)$ are introduced to respectively adopt a new goal or drop some old goals. We write *Bcap* and use it to denote the set of all belief update capabilities of an agent. *Bcap* thus does not include the two special actions for goal updating $\mathsf{adopt}(\phi)$ and $\mathsf{drop}(\phi)$. The set of all capabilities is then defined as $Cap = Bcap \cup \{\mathsf{adopt}(\phi), \mathsf{drop}(\phi) \mid \phi \in \mathcal{L}\}$. Individual capabilities are denoted by a.

The set of basic actions or capabilities associated with an agent determines what an agent *is able to do*. It does not specify *when* such a capability should be exercised and when performing a basic action is to the agent's advantage. To specify such conditions, the notion of a *conditional action* is introduced. A conditional action consists of a mental state condition expressed by a mental state formula and a basic action. The mental state condition of a conditional action states the conditions that must hold for the action to be selected. Conditional actions are denoted by the symbol b.

Definition 11.1.3 *(conditional action)*
A *conditional action* is a pair $\varphi \rightarrow do(\mathsf{a})$ such that $\varphi \in \mathcal{L}_M$ and $\mathsf{a} \in Cap$.

Informally, a conditional action $\varphi \rightarrow do(\mathsf{a})$ means that if the mental condition φ holds, then the agent may consider doing basic action a. Of course, if the mental state condition holds in the current state, the action a can only be successfully executed if the action is *enabled*, that is, only if its preconditions hold.

A GOAL agent consists of a specification of an *initial mental state* and a set of conditional actions.

Definition 11.1.4 *(GOAL agent)*
A *GOAL agent* is a triple $\langle \Pi, \sigma, \gamma \rangle$ where Π is a non-empty set of conditional actions, and $\langle \sigma, \gamma \rangle$ is a mental state.

11.1.1 The Operational Semantics of GOAL

One of the key ideas in the semantics of GOAL is to incorporate into the semantics a particular *commitment strategy* (cf. Rao & Georgeff (1990) and Cohen & Levesque (1990*a*)). The semantics is based on a particularly simple and transparent commitment strategy, called *blind commitment*. An agent that acts according to a blind commitment strategy drops a goal if and only if it believes that that goal has been achieved. By incorporating this commitment strategy into the semantics of GOAL, a *default* commitment strategy is built into agents. It is, however, only a default strategy and a programmer can overwrite this default strategy by means of the drop action. It is not possible, however, to adopt a goal ϕ in case the agent believes that ϕ is already achieved.

The semantics of action execution should now be defined in conformance with this basic commitment principle. Recall that the basic capabilities of an agent were interpreted as belief updates. Because of the default commitment strategy, there is a relation between beliefs and goals, however, and we should extend the belief update associated with a capability to a mental state transformer that updates beliefs as well as goals according to the blind commitment strategy. To get started, we thus assume that some specification of the belief update semantics of all capabilities - except for the two special actions adopt and drop which only update goals - is given. Our task is, then, to construct a mental state transformer semantics from this specification for each action. That is, we must specify how a basic action updates the complete current mental state of an agent starting with a specification of the belief update associated with the capability only.

From the default blind commitment strategy, we conclude that if a basic action a - different from an adopt or drop action - is executed, then a goal is dropped *only if* the agent *believes* that the goal has been accomplished after doing a. The revision of goals thus is based on the beliefs of the agent. The beliefs of an agent represent all the information that is available to an agent to decide whether or not to drop or adopt a goal. So, in case the agent believes that a goal has been achieved by performing some action, then this goal must be removed from the current goals of the agent. Besides the default commitment strategy, only the two special actions adopt and drop can result in a change to the goal base.

The initial specification of the belief updates associated with the capabilities $Bcap$ is formally represented by a partial function $\mathcal{T}$ of type : $Bcap \times \wp(\mathcal{L}) \to \wp(\mathcal{L})$. $\mathcal{T}(\mathsf{a}, \sigma)$ returns the result of updating belief base σ by performing action a. The fact that $\mathcal{T}$ is a *partial* function represents the fact that an action may not be *enabled* or executable in some belief states. The mental state transformer function $\mathcal{M}$ is derived from the semantic function $\mathcal{T}$ and also is a partial function. As explained, $\mathcal{M}(\mathsf{a}, \langle\sigma, \gamma\rangle)$ removes any goals from the goal base γ that have been achieved by doing a. The function $\mathcal{M}$ also defines the semantics of the two special actions adopt and drop. An $\mathsf{adopt}(\phi)$ action adds ϕ to the goal base if ϕ is consistent and ϕ is not believed to be the case. A $\mathsf{drop}(\phi)$ action removes every goal that entails ϕ from the goal base. As an example, consider the two extreme cases: $\mathsf{drop}(\mathsf{false})$ removes no goals, whereas $\mathsf{drop}(\mathsf{true})$ removes all current goals.

Definition 11.1.5 *(mental state transformer $\mathcal{M}$)*
Let $\langle\sigma, \gamma\rangle$ be a mental state, and $\mathcal{T}$ be a partial function that associates belief updates with agent capabilities. Then the partial function $\mathcal{M}$ is defined by:

$$\begin{array}{l}
\mathcal{M}(\mathsf{a}, \langle\sigma, \gamma\rangle) = \langle\mathcal{T}(\mathsf{a}, \sigma), \gamma \setminus \{\psi \in \gamma \mid \mathcal{T}(\mathsf{a}, \sigma) \models \psi\}\rangle \\
\qquad\qquad \text{for } \mathsf{a} \in Bcap \text{ if } \mathcal{T}(\mathsf{a}, \sigma) \text{ is defined,} \\
\mathcal{M}(\mathsf{a}, \langle\sigma, \gamma\rangle) \text{ is undefined for } \mathsf{a} \in Bcap \text{ if } \mathcal{T}(\mathsf{a}, \sigma) \text{ is undefined,} \\
\mathcal{M}(\mathsf{drop}(\phi), \langle\sigma, \gamma\rangle) = \langle\sigma, \gamma \setminus \{\psi \in \gamma \mid \psi \models \phi\}\rangle, \\
\mathcal{M}(\mathsf{adopt}(\phi), \langle\sigma, \gamma\rangle) = \langle\sigma, \gamma \cup \{\phi\}\rangle \text{ if } \sigma \not\models \phi \text{ and } \not\models \neg\phi, \\
\mathcal{M}(\mathsf{adopt}(\phi), \langle\sigma, \gamma\rangle) \text{ is undefined if } \sigma \models \phi \text{ or } \models \neg\phi.
\end{array}$$

The semantic function $\mathcal{M}$ maps an agent capability and a mental state to a new mental state. The capabilities of an agent are thus interpreted as *mental state transformers* by $\mathcal{M}$. Although it is not allowed to adopt a goal ϕ that is inconsistent - an $\mathsf{adopt}(\mathsf{false})$ is never enabled - there is no check on the global consistency of the goal base of an agent built into the semantics. This means that it is allowed to adopt a new goal which is inconsistent with another goal present in the goal base. For example, if the current goal base $\gamma = \{p\}$ contains p, it is legal to execute the action $\mathsf{adopt}(\neg p)$ resulting in a new goal base $\{p, \neg p\}$. Although inconsistent goals cannot be achieved at the same time, they may be achieved in some temporal order. Individual goals in the goal base, however, are required to be consistent. Thus, whereas local consistency is required (i.e.

individual goals must be consistent), global consistency of the goal base is not required (i.e. $\gamma = \{p, \neg p\}$ is a legal goal base).

The second idea incorporated into the semantics concerns the *selection of conditional actions.* A conditional action $\varphi \to do(\mathsf{a})$ may specify conditions on the beliefs as well as conditions on the goals of an agent. As is usual, conditions on the beliefs are taken as a precondition for action execution: only if the agent's current beliefs entail the belief conditions associated with φ the agent will select a for execution. The goal condition, however, is used in a different way. It is used as a means for the agent to determine whether or not the action will help bring about a particular goal of the agent. In short, the goal condition specifies where the action is good for. This does not mean that the action necessarily establishes the goal immediately, but rather may be taken as an indication that the action is helpful in bringing about a particular state of affairs. To make this discussion more precise, we introduce a formal definition of a *formula* ϕ *that partially fulfils a goal in a mental state* $\langle\sigma, \gamma\rangle$.

Definition 11.1.6 *(ϕ partially fulfils a goal in a mental state)*
Let $\langle\sigma, \gamma\rangle$ be a mental state, and $\phi \in \mathcal{L}$. Then:

$$\phi \leadsto_\sigma \gamma \text{ iff for some } \psi \in \gamma : \psi \models \phi \text{ and } \sigma \not\models \phi$$

Informally, the definition of $\phi \leadsto_\sigma \gamma$ can be paraphrased as follows: the agent needs to establish ϕ to realise one of its goals in γ, but does not believe that ϕ is the case. The formal definition of $\phi \leadsto_\sigma \gamma$ entails that the realisation of ϕ would bring about at least part of one of the goals in the goal base γ of the agent. The condition that ϕ is not entailed by the beliefs of the agent ensures that a goal is not a tautology. Of course, variations on this definition of the semantics of goals are conceivable. For example, one could propose a stronger definition of $\leadsto$ such that ϕ brings about the *complete* realisation of a goal in the current goal base γ instead of just part of such a goal. However, our definition of $\leadsto$ provides for a simple and clear principle for action selection: the action in a conditional action is only executed in case the goal condition associated with that action partially fulfils some goal in the current goal base of the agent.

The semantics of belief conditions $\mathsf{B}\phi$, goal conditions $\mathsf{G}\phi$ and mental state formulas is defined in terms of the consequence relation $\models$ and the 'partially fulfils relation' $\leadsto$.

Definition 11.1.7 *(semantics of mental state formulas)*
Let $\langle\sigma, \gamma\rangle$ be a mental state.

- $\langle\sigma, \gamma\rangle \models \mathsf{B}\phi$ iff $\sigma \models \phi$,
- $\langle\sigma, \gamma\rangle \models \mathsf{G}\psi$ iff $\psi \leadsto_\sigma \gamma$,
- $\langle\sigma, \gamma\rangle \models \neg\varphi$ iff $\langle\sigma, \gamma\rangle \not\models \varphi$,
- $\langle\sigma, \gamma\rangle \models \varphi_1 \wedge \varphi_2$ iff $\langle\sigma, \gamma\rangle \models \varphi_1$ and $\langle\sigma, \gamma\rangle \models \varphi_2$.

A number of properties of the belief and goal modalities and the relation between these operators are listed in the following lemma. By the necessitation rule, an agent believes all tautologies (Btrue). The first validity below states that the beliefs of an agent are consistent. The belief modality distributes over implication, which is expressed by the second validity. This implies that the beliefs of an agent are closed under logical consequence. The third validity is a consequence of the constraint on mental states and expresses that if an agent believes ϕ it does not have a goal to achieve ϕ. As a consequence, an agent cannot have a goal to achieve a tautology. An agent also does not have inconsistent goals, that is, $\neg\mathsf{G}(\mathsf{false})$ is valid.

The goal modality is a very weak logical operator. For example, the goal modality does not distribute over implication. A counter example is provided by the goal base $\gamma = \{p, p \to q\}$. Even $\mathsf{G}(\phi \wedge (\phi \to \psi)) \to \mathsf{G}\psi$ does not hold, because the agent may believe that ψ is the case even if it has a goal to achieve $\phi \wedge (\phi \to \psi)$. Because of the axiom $\mathsf{B}\psi \to \neg\mathsf{G}\psi$, we must have $\neg\mathsf{G}\psi$ in that case and we cannot conclude that $\mathsf{G}\psi$. From the fact that $\mathsf{G}\phi$ and $\mathsf{G}\psi$ hold, it is also not possible to conclude that $\mathsf{G}(\phi \wedge \psi)$. This reflects the fact that individual goals cannot be added to a single bigger goal; recall that two individual goals may be inconsistent ($\mathsf{G}\phi \wedge \mathsf{G}\neg\phi$ is satisfiable) in which case taking the conjunction would lead to an inconsistent goal. In sum, most of the usual problems that many logical operators for motivational attitudes suffer from do not apply to our G operator (cf. also Meyer et al. (1999)). Finally, the conditions that allow to conclude that the agent has a (sub)goal ψ are that the agent has a goal ϕ that logically entails ψ and that the agent does not believe that ψ is the case. The proof rule below then allows to conclude that $\mathsf{G}\psi$ holds.

Lemma 11.1.8

- $\models \phi \Rightarrow \models \mathsf{B}\phi$, for $\phi \in \mathcal{L}$,
- $\models \neg\mathsf{B}(\mathsf{false})$,
- $\models \mathsf{B}(\phi \to \psi) \to (\mathsf{B}\phi \to \mathsf{B}\psi)$,
- $\models \mathsf{B}\phi \to \neg\mathsf{G}\phi$,
- $\models \neg\mathsf{G}(\mathsf{true})$,
- $\models \neg\mathsf{G}(\mathsf{false})$,
- $\not\models \mathsf{G}(\phi \to \psi) \to (\mathsf{G}\phi \to \mathsf{G}\psi)$,
- $\not\models \mathsf{G}(\phi \wedge (\phi \to \psi)) \to \mathsf{G}\psi$,
- $\not\models (\mathsf{G}\phi \wedge \mathsf{G}\psi) \to \mathsf{G}(\phi \wedge \psi)$,
- $$\frac{\mathsf{G}\phi, \neg\mathsf{B}\psi, \models \phi \to \psi,}{\mathsf{G}\psi}$$

Now we have defined the formal semantics of mental state formulas, we are able to formally define the selection and execution of a conditional action. The selection of an action by an agent depends on the satisfaction conditions of the mental state condition associated with the action in a conditional action. The conditions for action selection thus may express conditions on both the belief and goal base of the agent. The belief conditions associated with the action formulate preconditions on the current belief base of the agent. Only if the current beliefs of the agent satisfy these conditions, an action may be selected. A condition $\mathsf{G}\phi$ on the goal base is satisfied if ϕ is entailed by one of the current goals of the agent (and thus, assuming the programmer did a good job, helps in bringing about one of these goals). The intuition here is that an agent is satisfied with anything bringing about at least (part of) one of its current goals. Note that a condition $\mathsf{G}\phi$ can only be satisfied if the agent does not already believe that ϕ is the case ($\sigma \not\models \phi$) which prevents an agent from performing an action without any need to do so.

In the definition below, we assume that the action component Π of an agent $\langle \Pi, \sigma, \gamma \rangle$ is fixed. The execution of an action gives rise to a *computation step* formally denoted by the transition relation $\xrightarrow{b}$ where b is the conditional action executed in the computation step. More than one computation step may be possible in a current state and the step relation $\longrightarrow$ thus denotes a *possible* computation step in a state. A computation step updates the current state and yields the next state of the computation. Note that because $\mathcal{M}$ is a partial function, a conditional action can only be successfully executed if both the condition is satisfied and the basic action is enabled.

Definition 11.1.9 *(action selection)*
Let $\langle \sigma, \gamma \rangle$ be a mental state and $b = \varphi \rightarrow do(\mathsf{a}) \in \Pi$. Then, as a rule, we have: If

- the mental condition φ holds in $\langle \sigma, \gamma \rangle$, i.e. $\langle \sigma, \gamma \rangle \models \varphi$, and
- a is enabled in $\langle \sigma, \gamma \rangle$, i.e. $\mathcal{M}(\mathsf{a}, \langle \sigma, \gamma \rangle)$ is defined,

then $\langle \sigma, \gamma \rangle \xrightarrow{b} \mathcal{M}(\mathsf{a}, \langle \sigma, \gamma \rangle)$ is a possible computation step. The relation $\longrightarrow$ is the smallest relation closed under this rule.

We say that a capability $\mathsf{a} \in \mathit{Cap}$ is *enabled* in a mental state $\langle \sigma, \gamma \rangle$ in case $\mathcal{M}(\mathsf{a}, \langle \sigma, \gamma \rangle)$ is defined. This definition implies that a belief update capability $\mathsf{a} \in \mathit{Bcap}$ is enabled if $\mathcal{T}(\mathsf{a}, \sigma)$ is defined. A conditional action b is *enabled* in a mental state $\langle \sigma, \gamma \rangle$ if there are σ', γ' such that $\langle \sigma, \gamma \rangle \xrightarrow{b} \langle \sigma', \gamma' \rangle$. Note that if a capability a is not enabled, a conditional action $\varphi \rightarrow do(\mathsf{a})$ is also not enabled. The special predicate *enabled* is introduced to denote that a capability a or conditional action b is enabled (denoted by $\mathit{enabled}(\mathsf{a})$ respectively $\mathit{enabled}(b)$).

Definition 11.1.10 *(semantics of enabled)*

- $\langle \sigma, \gamma \rangle \models \mathit{enabled}(\mathsf{a})$ iff $\mathcal{M}(\mathsf{a}, \langle \sigma, \gamma \rangle)$ is defined for $\mathsf{a} \in \mathit{Cap}$,

- $\langle\sigma,\gamma\rangle \models \mathit{enabled}(b)$ iff there are σ',γ' such that $\langle\sigma,\gamma\rangle \xrightarrow{b} \langle\sigma',\gamma'\rangle$ for conditional actions where $b = \varphi \rightarrow \mathit{do}(\mathsf{a})$.

The relation between the enabledness of capabilities and conditional actions is stated in the next lemma together with the fact that $\mathsf{drop}(\phi)$ is always enabled and a proof rule for deriving $\mathit{enabled}(\mathsf{adopt}(\phi))$.

Lemma 11.1.11

- $\models \mathit{enabled}(\varphi \rightarrow \mathit{do}(\mathsf{a})) \leftrightarrow (\varphi \wedge \mathit{enabled}(\mathsf{a}))$,
- $\models \mathit{enabled}(\mathsf{drop}(\phi))$,
- $\models \mathit{enabled}(\mathsf{adopt}(\phi)) \rightarrow \neg\mathsf{B}\phi$,
- $\dfrac{\not\models \neg\phi}{\neg\mathsf{B}\phi \rightarrow \mathit{enabled}(\mathsf{adopt}(\phi))}$

11.2 A Personal Assistant Example

In this section, we give an example to show how the programming language GOAL can be used to program agents. The example concerns a shopping agent that is able to buy books on the Internet on behalf of the user. The example provides for a simple illustration of how the programming language works. The agent in our example uses a standard procedure for buying a book. It first goes to a bookstore, in our case Amazon.com. At the web site of Amazon.com it searches for a particular book, and if the relevant page with the book details shows up, the agent puts the book in its shopping cart. In case the shopping cart of the agent contains some items, it is allowed to buy the items on behalf of the user. The idea is that the agent adopts a goal to buy a book if the user instructs it to do so.

The set of capabilities *Bcap* of the agent is defined by

$$\{\mathit{goto\_website}(\mathit{site}), \mathit{search}(\mathit{book}), \mathit{put\_in\_shopping\_cart}(\mathit{book}), \mathit{pay\_cart}\}$$

The capability *goto_website*(*site*) goes to the selected web page *site*. In our example, relevant web pages are the home page of the user, the main page of Amazon.com, web pages with information about books to buy, and a web page that shows the current items in the shopping cart of the agent. The capability *search*(*book*) is an action that can be selected at the main page of Amazon.com and selects the web page with information about *book*. The action *put_in_shopping_cart*(*book*) can be selected on the page concerning *book* and puts *book* in the cart; a new web page called *ContentCart* shows up showing the content of the cart. Finally, in case the cart is not empty the action *pay_cart* can be selected to pay for the books in the cart.

In the program text below, we assume that *book* is a variable referring to the specifics of the book the user wants to buy (in the example, we use variables as a means for abbreviation; variables should be thought of as being instantiated with the relevant arguments in such a way that predicates with variables reduce to propositions). The initial beliefs of the agent are that the current web page is the home page of the user, and that it is not possible to be on two different web pages at the same time. We also assume that the user has provided the agent with the goals to buy *The Intentional Stance* by Daniel Dennett and *Intentions, Plans, and Practical Reason* by Michael Bratman.

$$
\boxed{\begin{array}{l}
\Pi = \{ \\
\quad \mathsf{B}(current\_website(homepage(user)) \vee current\_website(ContentCart)) \\
\qquad \wedge \mathsf{G}(bought(book)) \rightarrow do(goto\_website(Amazon.com)), \\
\quad \mathsf{B}(current\_website(Amazon.com)) \wedge \neg \mathsf{B}(in\_cart(book)) \wedge \\
\qquad \mathsf{G}(bought(book)) \rightarrow do(search(book)), \\
\quad \mathsf{B}(current\_website(book)) \wedge \mathsf{G}(bought(book)) \rightarrow \\
\qquad do(put\_in\_shopping\_cart(book)), \\
\quad \mathsf{B}(in\_cart(book)) \wedge \mathsf{G}(bought(book)) \rightarrow do(pay\_cart)\}, \\
\sigma_0 = \{current\_webpage(homepage(user)), \\
\qquad \forall\, s, s'((s \neq s' \wedge current\_webpage(s)) \rightarrow \neg current\_webpage(s')) \\
\quad \}, \\
\gamma_0 = \{bought(\text{The Intentional Stance}) \wedge \\
\quad bought(\text{Intentions, Plans and Practical Reason})\}
\end{array}}
$$

GOAL Shopping Agent

Some of the details of this program will be discussed in the next chapter, when we prove some properties of the program. The agent basically follows the recipe for buying a book outlined above. For now, however, just note that the program is quite flexible, even though the agent more or less executes a fixed recipe for buying a book. The flexibility results from the agent's knowledge state and the non-determinism of the program. In particular, the ordering in which the actions are performed by the agent - which book to find first, buy a book one at a time or both in the same shopping cart, etc. is not determined by the program. The scheduling of these actions thus is not fixed by the program, and might be fixed arbitrarily on a particular agent architecture used to run the program.

11.3 Possible Extensions of GOAL

Although the basic features of the language GOAL are quite simple, the programming language GOAL is already quite powerful and can be used to program real agents. One of the restrictions of GOAL is that it only allows the use of basic actions. There are, however, several strategies to deal with this restriction. First of all, if a GOAL agent is proven correct, *any* scheduling of the basic actions *that is weakly fair* can be used to execute the agent.[1] More specifically,

[1]The notion of weak fairness has been defined in chapter 4 and in the next chapter a more extensive discussion and another definition of this notion can be found.

an interesting possibility is to define a mapping from GOAL agents to a particular agent architecture (cf. also Chandy & Misra (1988)). Such a mapping then may focus more on concerns like the efficiency or flexibility to determine the specific mapping that is most useful with respect to available architectures. There is only one condition that an agent architecture onto which a GOAL agent is mapped should obey, namely, it should implement a *weakly fair scheduling policy*. The concept of fair scheduling was defined in chapter 4 and is explained again in the next chapter.

A second strategy to circumvent to some extent the restriction that only basic actions are to be used is to use different grains of atomicity of basic actions. If a coarse-grained atomicity of basic actions is feasible for an application, one might consider taking complex plans as atomic actions and instantiate the basic actions in GOAL with these plans (however, termination of these complex plans should be guaranteed). Finally, in future research the extension of GOAL with a richer notion of action structure like for example plans could be explored. This would make the programming language more practical. The addition of such a richer notion, however, is not straightforward. At a minimum, more bookkeeping seems to be required to keep track of the goals that an agent already has chosen a plan for and which it is currently executing. This bookkeeping is needed, for example, to prevent the selection of more than one plan to achieve the same goal. Note that this problem was dealt with in GOAL by the immediate and complete execution of a selected action. It is therefore not yet clear how to give a semantics to a variant of GOAL extended with complex plans. The ideal, however, would be to combine the language GOAL which includes declarative goals with the agent programming language 3APL which includes planning features into a single new programming framework.

Apart from introducing more complex action structures, it would also be particularly interesting to extend GOAL with high-level communication primitives. Because both declarative knowledge as well as declarative goals are present in GOAL, communication primitives could be defined in the spirit of speech act theory (Searle 1969). The semantics of, for example, a request primitive could then be formally defined in terms of the knowledge and goals of an agent. Moreover, such a semantics would have a computational interpretation because both beliefs and goals have a computational interpretation in our framework.

11.4 Conclusion

Although a programming language dedicated to agent programming is not the only viable approach to building agents, we believe it is one of the more practical approaches for developing agents. Several other approaches to the design and implementation of agents have been proposed. One such approach promotes the use of *agent logics* for the specification of agent systems and aims at a further refinement of such specifications by means of an associated design methodology for the particular logic in use to implementations which meet this specification in, for example, an object-oriented programming language like Java. In this approach, there is no requirement on the existence of a natural mapping relating the end result of this development process - a Java implementation - and the formal specification in the logic. It is, however, not very clear how to implement these ideas for agent logics incorporating both informational and motivational attitudes and some researchers seem to have concluded from this that the notion of a motivational attitude (like a goal) is less useful than hoped for. Still another approach consists in the construction of *agent architectures* which 'implement' the different mental concepts. Such an architecture provides a template which can be instantiated with the relevant beliefs, goals, etc. Although this second approach is more practical than the first one, our main problem with this approach is that the architectures proposed so far tend to be quite complex. As a consequence, it is quite difficult to understand what behaviour an architecture that is instantiated will generate.

The design of a programming framework for intelligent agents resulted in the first part of this thesis in the programming language 3APL. 3APL supports the construction of intelligent agents, and reflects in a natural way the intentional concepts used to design agents. 3APL is a very powerful language, as was also shown in part II. It allows the construction of a multi-agent system of agents that communicate their beliefs, requests, perform actions, and construct plans to achieve their goals. A feature that distinguishes 3APL from most other agent frameworks is the reflective capabilities of 3APL agents due to practical reasoning rules.

However, 3APL includes a procedural instead of a declarative notion of goal. In this respect, it is similar to most other agent programming frameworks. It has been our aim in this chapter to show that it is feasible to incorporate declarative goals into a programming framework. The language GOAL was defined with a semantics that is computational and rather straightforward to implement, although this may require some restrictions on the logical reasoning involved on the part of GOAL agents. Because of the lack of a declarative concept of goal in languages like 3APL, it is hard to link these programming frameworks to agent logics. The use of such a link is obvious. It allows agent logics to be used to specify and verify the agents programmed in an agent language. In the next chapter, we show that such a link can be established for GOAL.

CHAPTER 12

Temporal Logic for GOAL

On top of the language GOAL and its semantics, we introduce a temporal logic to prove properties of GOAL agents. The logic extends the mental state language $\mathcal{L}_M$ of the previous chapter with features to reason about (conditional) actions and temporal properties of an agent. Hoare triples and associated proof rules are used to prove properties of actions. A temporal logic is introduced to reason about temporal aspects of GOAL agents. The logic is similar to other temporal logics but its semantics is derived from the operational semantics for GOAL.

First, we introduce the semantics for GOAL agents. Then we discuss basic action theories and in particular the use of Hoare triples for the specification of actions performed by GOAL agents. These Hoare triples play an important role in the programming logic since it can be shown that temporal properties of agents can be proven by means of proving Hoare triples for actions only. Then the language for expressing temporal properties and its semantics is defined and the fact that certain classes of interesting temporal properties can be reduced to properties of actions, expressed by Hoare triples, is proven. Finally, the example shopping agent of the previous chapter is proven correct by using the programming logic.

12.1 Semantics of GOAL Agents

The semantics of GOAL agents is derived directly from the operational semantics and the computation step relation $\longrightarrow$ as defined in the previous chapter. The meaning of a GOAL agent consists of a set of so called *traces*. A trace is an infinite computation sequence of consecutive mental states interleaved with the actions that are scheduled for execution in each of those mental states. The fact that a conditional action is scheduled for execution in a trace does not mean that it is also enabled in the particular state for which it has been scheduled. In case an action is scheduled but not enabled, the action is simply skipped and the resulting state is the same as the state before.

Definition 12.1.1 *(trace)*
A trace s is an infinite sequence $s_0, b_0, s_1, b_1, s_2, \ldots$ such that s_i is a mental state, b_i is a conditional action, and for every i we have: $s_i \xrightarrow{b_i} s_{i+1}$, or b_i is not enabled in s_i and $s_i = s_{i+1}$.

An important assumption in the semantics for GOAL is a *fairness* assumption. Fairness assumptions concern the fair selection of actions during the execution of a program. In our case, we make a *weak fairness* assumption (Manna & Pnueli 1992).

A trace is weakly fair if it is not the case that an action is always enabled from some point in time on but is never selected for execution.

Weak Fairness

This weak fairness assumption is built into the semantics by imposing a constraint on traces. By definition, a *fair trace* is a trace in which each of the actions is scheduled infinitely often. In a fair trace, there always will be a future time point at which an action is scheduled (considered for execution) and by this scheduling policy a fair trace implements the weak fairness assumption. However, note that the fact that an action is scheduled does not mean that the action also is enabled (and therefore, the selection of the action may result in an idle step which does not change the state).

The meaning of a GOAL agent now is defined as the set of fair traces in which the initial state is the initial mental state of the agent and each of the steps in the trace corresponds to the execution of a conditional action or an idle transition.

Definition 12.1.2 *(meaning of a GOAL agent)*
The meaning of a GOAL agent $\langle \Pi, \sigma_0, \gamma_0 \rangle$ is the set of *fair* traces S such that for $s \in S$ we have $s_0 = \langle \sigma_0, \gamma_0 \rangle$.

12.2 Hoare Triples

The specification of basic actions provides the basis for the programming logic, and, as we will show below, is all we need to prove properties of agents. Because they play such an important role in the proof theory of GOAL, the specification of the basic agent capabilities requires special care. In the proof theory of GOAL, Hoare triples of the form $\{\varphi\}\ b\ \{\psi\}$, where φ and ψ are *mental state formulas*, are used to specify actions. The use of Hoare triples in a formal treatment of traditional assignments is well-understood (Andrews 1991). Because the agent capabilities of GOAL agents are quite different from assignment actions, however, the traditional predicate transformer semantics is not applicable. GOAL agent capabilities are mental state transformers and, therefore, we require more extensive basic action theories to formally capture the effects of such actions. Hoare triples are used to specify the *postconditions* and the *frame conditions* of actions. The postconditions of an action specify the effects of an action whereas the frame conditions specify what is not changed by the action. Axioms for the predicate *enabled* specify the preconditions of actions.

The formal semantics of a Hoare triple for conditional actions is derived from the semantics of a GOAL agent and is defined relative to the set of traces S_A associated with the GOAL agent A. A Hoare triple for conditional actions thus expresses a property of an agent and not just a property of an action. The semantics of the basic capabilities are assumed to be fixed, however, and are not defined relative to an agent.

Definition 12.2.1 *(semantics of Hoare triples for basic actions)*
A *Hoare triple for basic capabilities* $\{\varphi\}$ a $\{\psi\}$ means that for all σ, γ

- $\langle\sigma,\gamma\rangle \models \varphi \wedge \mathit{enabled}(\mathsf{a}) \Rightarrow \mathcal{M}(\mathsf{a}, \langle\sigma,\gamma\rangle) \models \psi$, and
- $\langle\sigma,\gamma\rangle \models \varphi \wedge \neg \mathit{enabled}(\mathsf{a}) \Rightarrow \langle\sigma,\gamma\rangle \models \psi$.

To explain this definition, note that we made a case distinction between states in which the basic action is enabled and in which it is not enabled. In case the action is enabled, the postcondition ψ of the Hoare triple $\{\varphi\}$ a $\{\psi\}$ should be evaluated in the next state resulting from executing action a. In case the action is not enabled, however, the postcondition should be evaluated in the same state because a failed attempt to execute action a is interpreted as an idle step in which nothing changes.

Hoare triples for conditional actions are interpreted *relative to the set of traces* associated with the GOAL agent of which the action is a part. Below, we write $\varphi[s_i]$ to denote that a mental state formula φ holds in state s_i.

Definition 12.2.2 *(semantics of Hoare triples for conditional actions)*
Given an agent A, a *Hoare triple for conditional actions* $\{\varphi\}$ b $\{\psi\}$ (for A) means that for all traces $s \in S_A$ and i, we have that

$$(\varphi[s_i] \wedge b = b_i \in s) \Rightarrow \psi[s_{i+1}]$$

where $b_i \in s$ means that action b_i is taken in state i of trace s.

Of course, there is a relation between the execution of basic actions and that of conditional actions, and therefore there also is a relation between the two types of Hoare triples. The following lemma makes this relation precise.

Lemma 12.2.3 Let A be a GOAL agent and S_A be the meaning of A. Suppose that we have $\{\varphi \wedge \psi\}$ a $\{\varphi'\}$ and $S_A \models (\varphi \wedge \neg\psi) \rightarrow \varphi'$. Then we also have $\{\varphi\}$ $\psi \rightarrow \mathit{do}(\mathsf{a})$ $\{\varphi'\}$.

Proof: We need to prove that $(\varphi[s_i] \wedge (\psi \to do(\mathsf{a})) = b_i \in s) \Rightarrow \varphi'[s_{i+1}]$. Therefore, assume $\varphi[s_i] \wedge (\psi \to do(\mathsf{a})) = b_i \in s$. Two cases need to be distinguished: The case that the condition ψ holds in s_i and the case that it does not hold in s_i. In the former case, because we have $\{\varphi \wedge \psi\}$ a $\{\varphi'\}$ we then know that $s_{i+1} \models \varphi'$. In the latter case, the conditional action is not executed and $s_{i+1} = s_i$. From $((\varphi \wedge \neg\psi) \to \varphi')[s_i]$, $\varphi[s_i]$ and $\neg\psi[s_i]$ it then follows that $\varphi'[s_{i+1}]$ since φ' is a state formula. □

The definition of Hoare triples presented here formalises a *total correctness property*. A Hoare triple $\{\varphi\}$ b $\{\psi\}$ ensures that if initially φ holds, then an attempt to execute b results in a successor state and in that state ψ holds. This is different from *partial correctness* where no claims about the termination of actions and the existence of successor states are made.

12.3 Basic Action Theories

A *basic action theory* specifies the effects of the basic capabilities of an agent. It specifies when an action is enabled, it specifies the effects of an action and what does not change when an action is executed. In this respect, a GOAL basic action theory is similar to basic action theories defined in chapter 10 that are specified in the situation calculus. The structure of GOAL basic action theories, however, is quite different from basic action theories used by ConGolog.

A basic action theory consists of axioms for the predicate *enabled* for each basic capability, Hoare triples that specify the effects of basic capabilities and Hoare triples that specify frame axioms associated with these capabilities. Since the belief update capabilities of an agent are not fixed by the language GOAL but are user-defined, the user should specify the axioms and Hoare triples for belief update capabilities. The special actions for goal updating adopt and drop are part of GOAL and a set of axioms and Hoare triples for these actions is specified below.

Because in this chapter, our concern is not with the specification of basic action theories in particular, but with providing a programming logic for agents in which such specifications can be plugged in, we only provide some example specifications of the capabilities defined in the personal assistant example that we need in the proof of correctness below.

First, we specify a set of axioms for each of our basic actions that state when that action is enabled. Below, we abbreviate the book titles of the example, and write T for *The Intentional Stance* and I for *Intentions, Plans, and Practical Reason*. Moreover, recall that variables are used to abbreviate and should in fact be instantiated with the appropriate parameters. In the shopping agent example, we then have:

$$\begin{aligned}
&\mathit{enabled}(\mathit{goto\_website}(\mathit{site})) \leftrightarrow \mathsf{true},\\
&\mathit{enabled}(\mathit{search}(\mathit{book})) \leftrightarrow \mathsf{B}(\mathit{current\_website}(\mathit{Amazon.com})),\\
&\mathit{enabled}(\mathit{put\_in\_shopping\_cart}(\mathit{book})) \leftrightarrow \mathsf{B}(\mathit{current\_website}(\mathit{book})),\\
&\mathit{enabled}(\mathit{pay\_cart}) \leftrightarrow ((\mathsf{B}\mathit{in\_cart}(T) \vee \mathsf{B}\mathit{in\_cart}(I))\\
&\qquad\qquad\qquad\qquad \wedge \mathsf{B}\mathit{current\_website}(\mathit{ContentCart})).
\end{aligned}$$

Second, we list a number of effect axioms that specify the effects of a capability in particular situations defined by the preconditions of the Hoare triple.

- The action *goto_website*(*site*) results in moving to the relevant web page: $\{\mathsf{true}\}\ \mathit{goto\_website}(\mathit{site})\ \{\mathsf{B}\mathit{current\_website}(\mathit{site})\}$,
- At Amazon.com, searching for a book results in finding a page with relevant information about the book:

  $$\begin{aligned}
  &\{\mathsf{B}\mathit{current\_website}(\mathit{Amazon.com})\}\\
  &\quad \mathit{search}(\mathit{book})\\
  &\{\mathsf{B}\mathit{current\_website}(\mathit{book})\}
  \end{aligned}$$

- On the page with information about a particular book, selecting the action *put_in_shopping_cart*(*book*) results in the book being put in the cart; also, a new web page appears on which the contents of the cart are listed:

  $$\begin{aligned}
  &\{\mathsf{B}\mathit{current\_website}(\mathit{book})\}\\
  &\quad \mathit{put\_in\_shopping\_cart}(\mathit{book})\\
  &\{\mathsf{B}(\mathit{in\_cart}(\mathit{book}) \wedge \mathit{current\_website}(\mathit{ContentCart}))\}
  \end{aligned}$$

- In case *book* is in the cart, and the current web page presents a list of all the books in the cart, the action *pay_cart* may be selected resulting in the buying of all listed books:

  $$\begin{aligned}
  &\{\mathsf{B}(\mathit{in\_cart}(\mathit{book}) \wedge \mathit{current\_website}(\mathit{ContentCart}))\}\\
  &\quad \mathit{pay\_cart}\\
  &\{\neg\mathsf{B}\mathit{in\_cart}(\mathit{book}) \wedge \mathsf{B}(\mathit{bought}(\mathit{book}) \wedge \mathit{current\_website}(\mathit{Amazon.com}))\}
  \end{aligned}$$

Finally, we need a number of frame axioms that specify which properties are not changed by each of the capabilities of the agent. For example, both the capabilities *goto_website*(*site*) and *search*(*book*) do not change any beliefs about *in_cart*. Thus we have, e.g.:

$$\begin{aligned}
&\{\mathsf{B}\mathit{in\_cart}(\mathit{book})\}\ \mathit{goto\_website}(\mathit{site})\ \{\mathsf{B}\mathit{in\_cart}(\mathit{book})\}\\
&\{\mathsf{B}\mathit{in\_cart}(\mathit{book})\}\ \mathit{search}(\mathit{book})\ \{\mathsf{B}\mathit{in\_cart}(\mathit{book})\}
\end{aligned}$$

It will be clear that we need more frame axioms than these two, and some of these will be specified below in the proof of the correctness of the shopping agent.

It is important to realise that the only Hoare triples that need to be specified for agent capabilities are Hoare triples that concern the effects upon the *beliefs* of the agent. Changes and persistence of (some) goals due to executing actions can be derived with the proof rules and axioms below that are specifically designed to reason about the effects of actions on goals.

A theory of the belief update capabilities and their effects on the beliefs of an agent must be complemented with a theory about the effects of actions upon the goals of an agent. Such a theory should capture both the effects of the default commitment strategy as well as give a formal specification of the the drop and adopt actions.

The default commitment strategy imposes a constraint on the persistence of goals. A goal persists if it is not the case that after doing a the goal is believed to be achieved. Only action $\mathsf{drop}(\phi)$ is allowed to overrule this constraint. Therefore, in case $\mathsf{a} \neq \mathsf{drop}(\phi)$, we have that $\{\mathsf{G}\phi\}\ \mathsf{a}\ \{\mathsf{B}\phi \vee \mathsf{G}\phi\}$. This Hoare triple precisely captures the default commitment strategy and states that after executing an action the agent either believes it has achieved ϕ or it still has the goal ϕ if ϕ was a goal initially. A similar Hoare triple can be given for the persistence of the absence of a goal. Formally, we have $\{\neg\mathsf{G}\phi\}\ b\ \{\neg\mathsf{B}\phi \vee \neg\mathsf{G}\phi\}$. This Hoare triple states that the absence of a goal ϕ persists, and in case it does not persist the agent does not believe ϕ (anymore). We do not need this Hoare triple as an axiom, however, since it is a direct consequence of the fact that $\mathsf{B}\phi \rightarrow \neg\mathsf{G}\phi$. Note that the stronger $\{\neg\mathsf{G}\phi\}\ b\ \{\neg\mathsf{G}\phi\}$ does not hold, even if $b \neq \varphi \rightarrow do(\mathsf{adopt}(\phi))$.

It thus may also be the case that an agent believed it achieved ϕ but after doing an action b it no longer believes this to be the case and adopts ϕ as a goal again. For example, if the goal base $\gamma = \{p \wedge q\}$ and the belief base $\sigma = \{p\}$, then the agent does not have a goal to achieve p because it already believes p to be the case; however, in case an action changes the belief base such that p no longer is believed, the agent has a goal to achieve p (again). This provides for a mechanism similar to that of maintenance goals.

The specification of the special actions drop and adopt involves a number of frame axioms and a number of proof rules. The frame axioms capture the fact that neither of these actions has any effect on the beliefs of an agent:

- $\{\mathsf{B}\phi\}\ \mathsf{adopt}(\psi)\ \{\mathsf{B}\phi\}$, $\{\neg\mathsf{B}\phi\}\ \mathsf{adopt}(\psi)\ \{\neg\mathsf{B}\phi\}$,
- $\{\mathsf{B}\phi\}\ \mathsf{drop}(\psi)\ \{\mathsf{B}\phi\}$, $\{\neg\mathsf{B}\phi\}\ \mathsf{drop}(\psi)\ \{\neg\mathsf{B}\phi\}$.

The proof rules for the actions adopt and drop capture the effects on the goals of an agent. For each action, we list proof rules for the adoption respectively the dropping of goals, and for the persistence of goals. An agent adopts a new goal ϕ in case the agent does not believe ϕ and ϕ is not a contradiction.

$$\frac{\not\models \neg\phi}{\{\neg\mathsf{B}\phi\}\ \mathsf{adopt}(\phi)\ \{\mathsf{G}\phi\}}$$

An adopt action does not remove any current goals of the agent. Any existing goals thus persist when adopt is executed. The persistence of the absence of goals is somewhat more complicated in the case of an adopt action. An $\mathsf{adopt}(\phi)$ action does not add a new goal ψ in case ψ is not entailed by ϕ or ψ is believed to be the case:

$$\frac{}{\{\mathsf{G}\psi\}\ \mathsf{adopt}(\phi)\ \{\mathsf{G}\psi\}} \qquad \frac{\not\models \psi \rightarrow \phi}{\{\neg\mathsf{G}\phi\}\ \mathsf{adopt}(\psi)\ \{\neg\mathsf{G}\phi\}} \qquad \frac{}{\{\mathsf{B}\psi\}\ \mathsf{adopt}(\phi)\ \{\neg\mathsf{G}\psi\}}$$

A drop action $\mathsf{drop}(\phi)$ results in the removal of all goals that entail ϕ. This is captured by the proof rule:

$$\frac{\models \psi \rightarrow \phi}{\{\mathsf{G}\psi\}\ \mathsf{drop}(\phi)\ \{\neg\mathsf{G}\psi\}}$$

A drop action $\mathsf{drop}(\phi)$ never results in the adoption of new goals. The absence of a goal ψ thus persists when a drop action is executed. It is more difficult to formalise the persistence of a goal with respect to a drop action. Since a drop action $\mathsf{drop}(\phi)$ removes goals which entail ϕ, to conclude that a goal ψ persists after executing the action, we must make sure that the goal does not depend on a goal (is a subgoal) that is removed by the drop action. In case the conjunction $\phi \wedge \psi$ is not a goal, we know this for certain.

$$\frac{}{\{\neg\mathsf{G}\phi\}\ \mathsf{drop}(\psi)\ \{\neg\mathsf{G}\phi\}} \qquad \frac{}{\{\neg\mathsf{G}(\phi \wedge \psi) \wedge \mathsf{G}\phi\}\ \mathsf{drop}(\psi)\ \{\mathsf{G}\phi\}}$$

The basic action theories for GOAL include a number of proof rules to derive Hoare triples. The Rule for Infeasible Capabilities allows to derive frame axioms for a capability in case it is not enabled in a particular situation. The Rule for Conditional Actions allows the derivation of Hoare triples for conditional actions from Hoare triples for capabilities. This rule is justified by lemma 12.2.3. Finally, there are three rules for combining Hoare triples and for strengthening the precondition and weakening the postcondition.

Rule for Infeasible Capabilities:

$$\frac{\varphi \rightarrow \neg enabled(\mathsf{a})}{\{\varphi\}\ \mathsf{a}\ \{\varphi\}}$$

Rule for Conditional Actions:

$$\frac{\{\varphi \wedge \psi\}\ \mathsf{a}\ \{\varphi'\}, (\varphi \wedge \neg\psi) \rightarrow \varphi'}{\{\varphi\}\ \psi \rightarrow do(\mathsf{a})\ \{\varphi'\}}$$

Consequence Rule:

$$\frac{\varphi' \rightarrow \varphi, \{\varphi\}\ \mathsf{a}\ \{\psi\}, \psi \rightarrow \psi'}{\{\varphi'\}\ \mathsf{a}\ \{\psi'\}}$$

Conjunction Rule:

$$\frac{\{\varphi_1\}\ b\ \{\psi_1\}, \{\varphi_2\}\ b\ \{\psi_2\}}{\{\varphi_1 \wedge \varphi_2\}\ b\ \{\psi_1 \wedge \psi_2\}}$$

Disjunction Rule:

$$\frac{\{\varphi_1\}\ b\ \{\psi\}, \{\varphi_2\}\ b\ \{\psi\}}{\{\varphi_1 \vee \varphi_2\}\ b\ \{\psi\}}$$

12.4 Temporal logic

On top of the Hoare triples for specifying actions, a temporal logic is used to specify and verify properties of GOAL agents. Two new operators are introduced. The proposition **init** states that the agent is at the beginning of execution and nothing has happened yet. The second operator **until** is a weak until operator. φ **until** ψ means that ψ eventually becomes true and φ is true until ψ becomes true, or ψ never becomes true and φ remains true forever. Recall that $\mathcal{L}$ is a propositional language.

Definition 12.4.1 *(language of temporal logic $\mathcal{L}_T$ based on $\mathcal{L}$)*
The *temporal logic language* $\mathcal{L}_T$ is inductively defined by:

- **init** $\in \mathcal{L}_T$,
- $\mathit{enabled}(\mathsf{a}), \mathit{enabled}(\varphi \to \mathit{do}(\mathsf{a})) \in \mathcal{L}_T$ for $\mathsf{a} \in \mathit{Cap}$,
- if $\phi \in \mathcal{L}$, then $\mathsf{B}\phi, \mathsf{G}\phi \in \mathcal{L}_T$,
- if $\varphi, \psi \in \mathcal{L}_T$, then $\neg\varphi, \varphi \wedge \psi \in \mathcal{L}_T$,
- if $\varphi, \psi \in \mathcal{L}_T$, then φ **until** $\psi \in \mathcal{L}_T$.

A number of other well known temporal operators can be defined in terms of the operator **until** . The *always* operator $\Box\varphi$ is an abbreviation for φ **until** false, and the *eventuality* operator $\diamond\varphi$ is defined as $\neg\Box\neg\varphi$ as usual.

Temporal formulas are evaluated with respect to a trace s and a time point i. State formulas like $\mathsf{B}\phi$, $\mathsf{G}\psi$, $\mathit{enabled}(\mathsf{a})$ etc. are evaluated with respect to mental states.

Definition 12.4.2 *(semantics of temporal formulas)*
Let s be a trace and i be a natural number.

- $s, i \models$ **init** iff $i = 0$,
- $s, i \models \mathit{enabled}(\mathsf{a})$ iff $\mathit{enabled}(\mathsf{a})[s_i]$,
- $s, i \models \mathit{enabled}(\varphi \to \mathit{do}(\mathsf{a}))$ iff $\mathit{enabled}(\varphi \to \mathit{do}(\mathsf{a}))[s_i]$,
- $s, i \models \mathsf{B}\phi$ iff $\mathsf{B}\phi[s_i]$,
- $s, i \models \mathsf{G}\phi$ iff $\mathsf{G}\phi[s_i]$,
- $s, i \models \neg\varphi$ iff $s, i \not\models \varphi$,
- $s, i \models \varphi \wedge \psi$ iff $s, i \models \varphi$ and $s, i \models \psi$,
- $s, i \models \varphi$ **until** ψ iff
  $\exists j \geq i(s, j \models \psi \wedge \forall k(i \leq k < j(s, k \models \varphi)))$ or $\forall k \geq i(s, k \models \varphi)$.

We are particularly interested in temporal formulas that are valid with respect to the set of traces S_A associated with a GOAL agent A. Temporal formulas valid with respect to S_A express properties of the agent A.

Definition 12.4.3 Let S be a set of traces.

- $S \models \varphi$ iff $\forall\, s \in S, i(s, i \models \varphi)$,
- $\models \varphi$ iff $S \models \varphi$ where S is the set of all traces.

In general, two important types of temporal properties can be distinguished. Temporal properties are divided into *liveness* and *safety* properties. Liveness properties concern the progress that a program makes and express that a (good) state eventually will be reached. Safety properties, on the other hand, express that some (bad) state will never be entered. In the rest of this section, we discuss a number of specific liveness and safety properties of an agent $A = \langle \Pi_A, \sigma, \gamma \rangle$.

We show that each of the properties that we discuss are equivalent to a set of Hoare triples. The importance of this result is that it shows that temporal properties of agents can be proven by inspection of the program text only. The fact that proofs of agent properties can be constructed by inspection of the program text means that there is no need to reason about individual traces of an agent or its operational behaviour. In general, reasoning about the program text is more economical since the number of traces associated with a program is exponential in the size of the program.

The first property we discuss concerns a safety property, and is expressed by the temporal formula $\varphi \rightarrow (\varphi\ \mathbf{until}\ \psi)$. Properties in this context always refer to agent properties and are evaluated with respect to the set of traces associated with that agent. Therefore, we can explain the informal meaning of the property as stating that if φ ever becomes true, then it remains true until ψ becomes true. By definition, we write this property as φ **unless** ψ:

$$\varphi\ \mathbf{unless}\ \psi \overset{df}{=} \varphi \rightarrow (\varphi\ \mathbf{until}\ \psi)$$

An important special case of an **unless** property is φ **unless** false, which expresses that if φ ever becomes true, it will remain true. φ **unless** false means that φ is a *stable* property of the agent. In case we also have **init** $\rightarrow \varphi$, where **init** denotes the initial starting point of execution, φ is always true and is an *invariant* of the program.

Now we show that **unless** properties of an agent $A = \langle \Pi, \sigma, \gamma \rangle$ are equivalent to a set of Hoare triples for basic actions in Π. This shows that we can prove **unless** properties by proving a set of Hoare triples. The proof relies on the fact that if we can prove that after executing any action from Π either φ persists or ψ becomes true, we can conclude that φ **unless** ψ.

Theorem 12.4.4 Let $A = \langle \Pi_A, \sigma, \gamma \rangle$. Then:

$$\forall\, b \in \Pi_A(\{\varphi \wedge \neg\psi\}\ b\ \{\varphi \vee \psi\}) \text{ iff } S_A \models \varphi\ \mathbf{unless}\ \psi$$

Proof: The proof from right to left is the easiest direction in the proof. Suppose that $S_A \models \varphi$ **unless** ψ and $s, i \models \varphi$. This implies that $s, i \models \varphi$ **until** ψ. In case we also have $s, i \models \psi$, we are done. So, assume $s, i \models \neg\psi$ and action b is selected in the trace at state s_i. From the semantics of **until** we then know that $\varphi \vee \psi$ holds at state s_{i+1}, and we immediately obtain $\{\varphi \wedge \neg\psi\}\ b\ \{\varphi \vee \psi\}$ since trace s and time point i were arbitrarily chosen.

We prove the left to right case by contraposition. Suppose that

$$(*)\ \forall\, b \in \Pi_A(\{\varphi \wedge \neg\psi\}\ b\ \{\varphi \vee \psi\})$$

and for some $s \in S_A$ we have $s, i \not\models \varphi$ **unless** ψ. The latter fact means that we have $s, i \models \varphi$ and $s, i \not\models \varphi$ **until** ψ. $s, i \not\models \varphi$ **until** ψ implies that either (i) ψ is never established at some $j \geq i$ but we do have $\neg\varphi$ at some time point $k > i$ or (ii) ψ is established at some time $j > i$, but in between i and any such j it is not always the case that φ holds.

In the first case (i), let $k > i$ be the smallest k such that $s, k \not\models \varphi$. Then, we have $s, k-1 \models \varphi \wedge \neg\psi$ and $s, k \models \neg\varphi \wedge \neg\psi$. In that case, we must have that in state s_{k-1} a conditional action has been performed since the state has been changed. From (*) we then derive a contradiction.

In the second case (ii), let $k > i$ be the smallest k such that $s, k \models \psi$. Then we know that there is a smallest j such that $i < j < k$ and $s, j \not\models \varphi$ ($j \neq i$ since $s, i \models \varphi$). This means that we have $s, j-1 \models \varphi \wedge \neg\psi$. In that case, we again must have that in state s_{j-1} a conditional action has been performed. From (*) we then derive a contradiction. □

Liveness properties involve eventualities which state that some state will be reached starting from a particular situation. To express a special class of such properties, we introduce the operator φ **ensures** ψ. φ **ensures** ψ informally means that φ guarantees the realisation of ψ, and is defined as:

$$\varphi\ \textbf{ensures}\ \psi \overset{df}{=} \varphi\ \textbf{unless}\ \psi \wedge (\varphi \rightarrow \diamond\psi)$$

φ **ensures** ψ thus ensures that ψ is eventually realised starting in a situation in which φ holds, and requires that φ holds until ψ is realised. For the class of **ensures** properties, we can again show that these properties can be proven by proving a set of Hoare triples. The proof of a **ensures** property thus can be reduced to the proof of a set of Hoare triples.

Theorem 12.4.5 Let $A = \langle \Pi_A, \sigma, \gamma \rangle$. Then:

$$\begin{aligned} &\forall\, b \in \Pi_A(\{\varphi \wedge \neg\psi\}\ b\ \{\varphi \vee \psi\}) \wedge \exists\, b \in \Pi_A(\{\varphi \wedge \neg\psi\}\ b\ \{\psi\}) \\ &\qquad \Rightarrow S_A \models \varphi\ \textbf{ensures}\ \psi \end{aligned}$$

Proof: In the proof, we need the weak fairness assumption. Since φ **ensures** ψ is defined as φ **unless** $\psi \wedge (\varphi \rightarrow \diamond\psi)$, by theorem 12.4.4 we only need to prove that $S_A \models \varphi \rightarrow \diamond\psi$ given that $\forall\, b \in \Pi_A(\{\varphi \wedge \neg\psi\}\ b\ \{\varphi \vee \psi\}) \wedge \exists\, b \in \Pi_A(\{\varphi \wedge \neg\psi\}\ b\ \{\psi\})$. Now suppose, to arrive at a contradiction, that for some time point i and trace $s \in S_A$ we have: $s, i \models \varphi \wedge \neg\psi$ and assume that for all later points $j > i$ we have $s, j \models \neg\psi$. In that case, we know that for all $j > i$ we have $s, j \models \varphi \wedge \neg\psi$ (because we may assume φ **unless** ψ). However, we also know that there is an action b that is enabled in a state in which $\varphi \wedge \neg\psi$ holds and transforms this state to a state in which ψ holds. The action b thus is always enabled, but apparently never taken. This is forbidden by weak fairness, and we arrive at a contradiction. $\Box$

The implication in the other direction in theorem 12.4.5 does not hold. A counterexample is provided by the program:

$$\begin{aligned}
&\Pi = \{\\
&\qquad \mathsf{B}(\neg p \wedge q) \rightarrow do(\mathsf{ins}(p)),\\
&\qquad \mathsf{B}(\neg p \wedge r) \rightarrow do(\mathsf{ins}(p)),\\
&\qquad \mathsf{B}p \rightarrow do(\mathsf{ins}(\neg p \wedge q)),\\
&\qquad \mathsf{B}p \rightarrow do(\mathsf{ins}(\neg p \wedge r))\},\\
&\sigma = \{p\},\\
&\gamma = \varnothing.
\end{aligned}$$

where

$$\begin{aligned}
&\mathcal{T}(\mathsf{ins}(p), \{\neg p \wedge q\}) = \{p \wedge q\}, \mathcal{T}(\mathsf{ins}(p), \{\neg p \wedge r\}) = \{p \wedge r\},\\
&\mathcal{T}(\mathsf{ins}(\neg p \wedge q), \{p\}) = \mathcal{T}(\mathsf{ins}(\neg p \wedge q), \{\{p \wedge q\}) = \{\neg p \wedge q\},\\
&\mathcal{T}(\mathsf{ins}(\neg p \wedge r), \{p\}) = \mathcal{T}(\mathsf{ins}(\neg p \wedge r), \{p \wedge r\}) = \{\neg p \wedge r\}.
\end{aligned}$$

For this program, we have that $\mathsf{B}\neg p$ **ensures** $\mathsf{B}p$ holds, but we do not have $\{\mathsf{B}\neg p \wedge \neg\mathsf{B}p\}\ b\ \{\mathsf{B}p\}$ for some $b \in \Pi$.

Finally, we introduce a third temporal operator 'leads to' $\mapsto$. The operator $\varphi \mapsto \psi$ differs from **ensures** in that it does not require φ to remain true until ψ is established, and is derived from the **ensures** operator. $\mapsto$ is defined as the transitive, disjunctive closure of **ensures** .

Definition 12.4.6 *(leads to operator)*
The leads to operator $\mapsto$ is defined by:

$$\boxed{\frac{\varphi\ \textbf{ensures}\ \psi}{\varphi \mapsto \psi} \qquad \frac{\varphi \mapsto \chi, \chi \mapsto \psi}{\varphi \mapsto \psi} \qquad \frac{\varphi_1 \mapsto \psi, \ldots, \varphi_n \mapsto \psi}{(\varphi_1 \vee \ldots \vee \varphi_n) \rightarrow \psi}}$$

The meaning of the 'leads to' operator is captured by the following lemma. $\varphi \mapsto \psi$ means that given φ condition ψ will eventually be realised. The proof of the lemma is an easy induction on the definition of $\mapsto$.

Lemma 12.4.7 $\varphi \mapsto \psi \models \varphi \rightarrow \Diamond\psi$.

12.5 Proving Agents Correct

In this section, we use the programming logic to prove the correctness of the example shopping agent of the previous chapter. We do not present all the details, but provide enough details to illustrate the use of the programming logic. Before we discuss what it means that an agent program is correct and provide a proof which shows that our example agent is correct, we introduce some notation. The notation involves a number of abbreviations concerning names and propositions in the language of our example agent:

- From now on, instead of the formula $current\_website(sitename)$ we simply write $sitename$; for example, we write $Amazon.com$ instead of the proposition $current\_website(Amazon.com)$,
- As before, the book titles *The Intentional Stance* and *Intentions, Plans and Practical Reason* that the agent intends to buy are abbreviated to T and I respectively. These conventions can result in formulas like $\mathsf{B}(T)$, which means that the agent is at the web page concerning the book *The Intentional Stance.*

A simple and intuitive correctness property, which is natural in this context and is applicable to our example agent, states that a GOAL agent is *correct* when the agent program realises the initial goals of the agent. For this subclass of correctness properties, we may consider the agent to be finished upon establishing the initial goals and in that case the agent could be terminated. Of course, it is also possible to continue the execution of such agents. This class of correctness properties can be expressed by means of temporal formulas like $\mathsf{G}\phi \to \diamond\neg\mathsf{G}\phi$. Other correctness properties are conceivable, of course, but not all of them can be expressed in the temporal proof logic for GOAL.

12.6 Correctness of the Shopping Agent

From the discussion above, we conclude that the interesting property to prove for our example program is the following property:

$$Bcond \wedge \mathsf{G}(bought(T) \wedge bought(I)) \mapsto \mathsf{B}(bought(T) \wedge bought(I))$$

where $Bcond$ is a condition on the initial beliefs of the agent. More specifically, $Bcond$ is defined by:

$$\begin{array}{l} \mathsf{B}current\_webpage(homepage(user)) \wedge \neg\mathsf{B}in\_cart(T) \wedge \neg\mathsf{B}in\_cart(I) \wedge \\ \mathsf{B}(\forall\, s, s'((s \neq s' \wedge current\_webpage(s)) \to \neg current\_webpage(s'))) \end{array}$$

The correctness property states that the goal to buy the books *The Intentional Stance* and *Intentions, Plans and Practical Reason*, given some initial conditions on the beliefs of the agent, leads to buying (or believing to have bought) these books. Note that this property expresses a *total correctness* property. It states both that the program behaves as desired and that it will eventually reach the desired goal state. Another reason for considering this property to express the correctness of our example agent is the fact that in case the goals mentioned are achieved, the agent believes that they are achieved once and for all. That is, the agent will never adopt the same goal again once it is achieved, and, since the agent will not adopt new goals, once each of the goals has been achieved the agent has no goals left to achieve and is done.

12.7 Invariants and Frame Axioms

To be able to prove correctness, we need a number of frame axioms. There is a close relation between frame axioms and invariants of a program. Frame axioms state which properties are not changed by (particular) actions. We call a property *stable* in case that property once it becomes true, remains true whatever action is performed. The fact that a property p is stable is expressed by the formula p **unless** false. In case a stable property holds initially, the property is called an *invariant* of the program. Invariant properties always hold during the execution of an agent. We need frame axioms to prove that a property is stable or invariant.

For our example program, we have an invariant that states that we cannot view two web pages at the same time:

$$inv = \mathsf{B}\,\forall\, s, s'((s \neq s' \wedge current\_webpage(s)) \rightarrow \neg current\_webpage(s'))$$

To prove that inv is an invariant of the agent, we need frame axioms stating that when inv holds before the execution of an action it still holds after executing that action. Formally, for each $\mathsf{a} \in Cap$, we need: $\{inv\}\ \mathsf{a}\ \{inv\}$. These frame axioms need to be specified by the user, and for our example agent we assume that they are indeed true. By means of the Consequence Rule (strengthen the precondition of the Hoare triples for capabilities a) and the Rule for Conditional Actions (instantiate φ and φ' with inv), we then obtain that $\{inv\}\ b\ \{inv\}$ for all $b \in \Pi$. By theorem 12.4.4, we then know that inv **unless** false. Because we also have that initially inv holds since we know that $\langle\sigma_0, \gamma_0\rangle \models inv$ holds for our shopping agent, we may conclude that **init** $\rightarrow inv \wedge inv$ **unless** false. inv thus is an invariant and holds at all times during the execution of the agent. Because of this fact, we do not mention inv explicitly anymore in the proofs below, but will freely use the property when we need it.

A second property that is stable is the property $status(book)$:

$$status(book) \overset{df}{=} (\mathsf{B}in\_cart(book) \wedge \mathsf{G}bought(book)) \vee \mathsf{B}bought(book)$$

The fact that $status(book)$ is stable means that once a book is in the cart and it is a goal to buy the book, it remains in the cart and is only removed from the cart when it is bought.

The proof obligations to prove that $status(book)$ is a *stable* property, consist of supplying proofs for $\{status(book)\}\ b\ \{status(book)\}$ for each conditional action $b \in \Pi$ of the shopping agent (cf. theorem 12.4.4). By the Rule for Conditional Actions, therefore, it is sufficient to prove for each conditional action $\psi \rightarrow do(\mathsf{a}) \in \Pi$ that $\{status(book) \wedge \psi\}\ \mathsf{a}\ \{status(book)\}$ and $(status(book) \wedge \neg\psi) \rightarrow status(book)$. The latter implication is trivial. Moreover, it is clear that to prove the Hoare triples it is sufficient to prove $\{status(book)\}\ \mathsf{a}\ \{status(book)\}$ since we can strengthen the precondition by means of the Consequence Rule. The proof obligations thus reduce to proving $\{status(book)\}\ \mathsf{a}\ \{status(book)\}$ for each capability of the shopping agent.

Again, we cannot prove these Hoare triples without a number of frame axioms. Because no capability is allowed to reverse the fact that a book has been bought, for each capability, we can specify a frame axiom for the predicate *bought*:

$$(1) \qquad \{\mathsf{B}bought(book)\}\ \mathsf{a}\ \{\mathsf{B}bought(book)\}$$

In case the book is not yet bought, selecting action *pay_cart* may change the contents of the cart and therefore we first treat the other three actions *goto_website*, *search*, and *put_in_shopping_cart* which are not supposed to change the contents of the cart. For each of the latter three capabilities we therefore add the frame axioms:

$$\{\mathsf{B}in\_cart(book) \wedge \neg\mathsf{B}bought(book)\}\ \mathsf{a}\ \{\mathsf{B}in\_cart(book) \wedge \neg\mathsf{B}bought(book)\}$$

where $\mathsf{a} \neq pay\_cart$. Note that these frame axioms do not refer to goals but only refer to the beliefs of the agent, in agreement with our claim that only Hoare triples for belief updates need to be specified by the user. By using the axiom $\mathsf{G}bought(book) \rightarrow \neg\mathsf{B}bought(book)$ and the Consequence Rule, however, we can conclude that:

$$\{\mathsf{B}in\_cart(book) \wedge \mathsf{G}bought(book)\}\ \mathsf{a}\ \{\mathsf{B}in\_cart(book) \wedge \neg\mathsf{B}bought(book)\}$$

By combining these Hoare triples with the axiom

$$\{\mathsf{G}bought(book)\}\ \mathsf{a}\ \{\mathsf{B}bought(book) \vee \mathsf{G}bought(book)\}$$

by means of the Conjunction Rule and by rewriting the postcondition with the Consequence Rule, we then obtain

$$(2) \qquad \begin{array}{c} \{\mathsf{B}in\_cart(book) \wedge \mathsf{G}bought(book)\} \\ \mathsf{a} \\ \{\mathsf{B}in\_cart(book) \wedge \mathsf{G}bought(book)\} \end{array}$$

where $\mathsf{a} \neq pay\_cart$. By weakening the postconditions of (1) and (2) by means of the Consequence Rule and combining the result with the Disjunction Rule, it is then possible to conclude that $\{status(book)\}\ \mathsf{a}\ \{status(book)\}$ for $\mathsf{a} \neq pay\_cart$.

As before, in the case of capability *pay_cart* we deal with each of the disjuncts of *status*(*book*) in turn. The second disjunct can be handled as before, but the first disjunct is more involved this time because *pay_cart* can change both the content of the cart and the goal to buy a book if it is enabled. Note that *pay_cart* only is enabled in case $\mathsf{B}\,ContentCart$ holds. In case $\mathsf{B}\,ContentCart$ holds and *pay_cart* is enabled, from the effect axiom for *pay_cart* and the Consequence Rule we obtain

$$(3) \qquad \begin{array}{c} \{\mathsf{B}in\_cart(book) \wedge \mathsf{G}bought(book) \wedge \mathsf{B}\,ContentCart\} \\ pay\_cart \\ \{\mathsf{B}bought(book)\} \end{array}$$

In case $\neg\mathsf{B}\,ContentCart$ holds and *pay_cart* is not enabled, we use the Rule for Infeasible Capabilities to conclude that

$$(4)\quad \begin{array}{c}\{\mathsf{B}in\_cart(book) \wedge \mathsf{G}bought(book) \wedge \neg\mathsf{B}\,ContentCart\}\\ pay\_cart\\ \{\mathsf{B}in\_cart(book) \wedge \mathsf{G}bought(book) \wedge \neg\mathsf{B}\,ContentCart\}\end{array}$$

By means of the Consequence Rule and the Disjunction Rule, we then can conclude from (1), (3) and (4) that $\{status(book)\}$ *pay_cart* $\{status(book)\}$, and we are done.

12.8 Proof Outline

The main proof steps to prove our agent example correct are listed next. The proof steps below consists of a number of **ensures** formulas which together prove that the program reaches its goal in a finite number of steps.

(1) $\mathsf{B}homepage(user) \wedge \neg\mathsf{B}in\_cart(T) \wedge \mathsf{G}bought(T) \wedge \neg\mathsf{B}in\_cart(I)$
$\wedge\mathsf{G}bought(I)$ **ensures**
$\mathsf{B}Amazon.com \wedge \neg\mathsf{B}in\_cart(T) \wedge \mathsf{G}bought(T) \wedge \neg\mathsf{B}in\_cart(I)$
$\wedge\mathsf{G}bought(I)$
(2) $\mathsf{B}Amazon.com \wedge \neg\mathsf{B}in\_cart(T) \wedge \mathsf{G}bought(T) \wedge \neg\mathsf{B}in\_cart(I)$
$\wedge\mathsf{G}bought(I)$ **ensures**
$[(\mathsf{B}(T) \wedge \mathsf{G}bought(T) \wedge \neg\mathsf{B}in\_cart(I) \wedge \mathsf{G}bought(I))\vee$
$(\mathsf{B}(I) \wedge \mathsf{G}bought(I) \wedge \neg\mathsf{B}in\_cart(T) \wedge \mathsf{G}bought(T))]$
(3) $\mathsf{B}(T) \wedge \mathsf{G}bought(T) \wedge \neg\mathsf{B}in\_cart(I) \wedge \mathsf{G}bought(I)$ **ensures**
$\mathsf{B}in\_cart(T) \wedge \mathsf{G}bought(T) \wedge \neg\mathsf{B}in\_cart(I) \wedge \mathsf{G}bought(I)$
$\wedge\mathsf{B}\,ContentCart$
(4) $\mathsf{B}in\_cart(T) \wedge \mathsf{G}bought(T) \wedge \neg\mathsf{B}in\_cart(I) \wedge \mathsf{G}bought(I)$ **ensures**
$\mathsf{B}Amazon.com \wedge \neg\mathsf{B}in\_cart(I) \wedge \mathsf{G}bought(I) \wedge status(T)$
(5) $\mathsf{B}(Amazon.com) \wedge \neg\mathsf{B}in\_cart(I) \wedge \mathsf{G}bought(I) \wedge status(T)$ **ensures**
$\mathsf{B}(I) \wedge \mathsf{G}bought(I) \wedge status(T)$
(6) $\mathsf{B}(I) \wedge \mathsf{G}bought(I) \wedge status(T)$ **ensures**
$\mathsf{B}in\_cart(I) \wedge \mathsf{G}bought(I) \wedge \mathsf{B}\,ContentCart \wedge status(T)$
(7) $\mathsf{B}in\_cart(I) \wedge \mathsf{G}bought(I) \wedge \mathsf{B}\,ContentCart \wedge status(T)$ **ensures**
$\mathsf{B}bought(T) \wedge \mathsf{B}bought(I)$

At step 3, the proof is split up into two subproofs, one for each of the disjuncts of the disjunct that is ensured in step 2. The proof for the other disjunct is completely analogous. By applying the rules for the 'leads to' operator the third to seventh step result in:

(a) $B(T) \wedge \mathsf{G}bought(T) \wedge \neg\mathsf{B}in\_cart(I) \wedge \mathsf{G}bought(I) \mapsto$
$\mathsf{B}bought(T) \wedge \mathsf{B}bought(I)$
(b) $B(I) \wedge \mathsf{G}bought(I) \wedge \neg\mathsf{B}in\_cart(T) \wedge \mathsf{G}bought(T) \mapsto$
$\mathsf{B}bought(T) \wedge \mathsf{B}bought(I)$

Combining (a) and (b) by the disjunction rule for the 'leads to' operator and by using the transitivity of 'leads to' we then obtain the desired correctness result:

$$\mathit{Bcond} \wedge \mathsf{G}(\mathit{bought}(T) \wedge \mathit{bought}(I)) \mapsto \mathsf{B}(\mathit{bought}(T) \wedge \mathit{bought}(I))$$

with *Bcond* as defined previously.

Step 1 We now discuss the first proof step in somewhat more detail. The remainder of the proof is left to the reader. The proof of a formula φ **ensures** ψ requires that we show that every action b in the Personal Assistant program satisfies the Hoare triple $\{\varphi \wedge \neg\psi\}\ b\ \{\varphi \vee \psi\}$ and that there is at least one action b' which satisfies the Hoare triple $\{\varphi \wedge \neg\psi\}\ b'\ \{\psi\}$. By inspection of the program, in our case the proof obligations turn out to be:

$$\begin{array}{l} \{\mathsf{B}\mathit{homepage}(\mathit{user}) \wedge \neg\mathsf{B}\mathit{in\_cart}(T) \wedge \mathsf{G}\mathit{bought}(T) \wedge \\ \qquad\qquad \neg\mathsf{B}\mathit{in\_cart}(I) \wedge \mathsf{G}\mathit{bought}(I)\} \\ \quad b \\ \{\mathsf{B}\mathit{homepage}(\mathit{user}) \wedge \neg\mathsf{B}\mathit{in\_cart}(T) \wedge \mathsf{G}\mathit{bought}(T) \wedge \\ \qquad\qquad \neg\mathsf{B}\mathit{in\_cart}(I) \wedge \mathsf{G}\mathit{bought}(I)\ \} \end{array}$$

where b is one of the actions

$$\begin{array}{l} \mathsf{B}(\mathit{Amazon.com}) \wedge \neg\mathsf{B}(\mathit{in\_cart}(\mathit{book})) \wedge \mathsf{G}(\mathit{bought}(\mathit{book})) \\ \qquad\qquad \rightarrow \mathit{do}(\mathit{search}(\mathit{book})), \\ \mathsf{B}(\mathit{book}) \wedge \mathsf{G}(\mathit{bought}(\mathit{book})) \rightarrow \mathit{do}(\mathit{put\_in\_shopping\_cart}(\mathit{book})), \\ \mathsf{B}(\mathit{in\_cart}(\mathit{book})) \wedge \mathsf{G}(\mathit{bought}(\mathit{book})) \rightarrow \mathit{do}(\mathit{pay\_cart})\} \end{array}$$

and

$$\begin{array}{l} \{\mathsf{B}\mathit{homepage}(\mathit{user}) \wedge \neg\mathsf{B}\mathit{in\_cart}(T) \wedge \mathsf{G}\mathit{bought}(T) \wedge \\ \qquad\qquad \neg\mathsf{B}\mathit{in\_cart}(I) \wedge \mathsf{G}\mathit{bought}(I) \\ \ \} \\ \quad \mathsf{B}(\mathit{homepage}(\mathit{user}) \vee \mathit{ContentCart}) \wedge \mathsf{G}(\mathit{bought}(\mathit{book})) \\ \qquad\qquad \rightarrow \mathit{do}(\mathit{goto\_website}(\mathit{Amazon.com})) \\ \{\mathsf{B}\mathit{Amazon.com} \wedge \neg\mathsf{B}\mathit{in\_cart}(T) \wedge \mathsf{G}\mathit{bought}(T) \wedge \\ \qquad\qquad \neg\mathsf{B}\mathit{in\_cart}(I) \wedge \mathsf{G}\mathit{bought}(I) \\ \ \} \end{array}$$

The proofs of the first three Hoare triples are derived by using the Rule for Conditional Actions. The key point is noticing that each of the conditions of the conditional actions involved refers to a web page different from the web page *homepage*(*user*) referred to in the precondition of the Hoare triple. The proof thus consists of using the fact that initially $\mathsf{B}\mathit{homepage}(\mathit{user})$ and the invariant *inv* to derive an inconsistency which immediately yield the desired Hoare triples by means of the Rule for Conditional Actions.

To prove the Hoare triple for

$$\begin{array}{l} \mathsf{B}(\mathit{homepage}(\mathit{user}) \vee \mathit{ContentCart}) \wedge \mathsf{G}(\mathit{bought}(\mathit{book})) \\ \qquad\qquad \rightarrow \mathit{do}(\mathit{goto\_website}(\mathit{Amazon.com})) \end{array}$$

we use the effect axiom (5) for *goto_website* and the frame axiom (6):

$$\begin{array}{ll} & \{\mathsf{B}\,homepage(user)\} \\ (5) & \quad goto\_website(Amazon.com) \\ & \{\mathsf{B}\,Amazon.com\} \\ & \text{and} \\ & \{\neg\mathsf{B}\,in\_cart(book) \wedge \neg\mathsf{B}\,bought(book)\} \\ (6) & \quad goto\_website(Amazon.com) \\ & \{\neg\mathsf{B}\,in\_cart(book) \wedge \neg\mathsf{B}\,bought(book)\} \end{array}$$

By using the axiom

$$\begin{array}{l} \{\mathsf{G}\,bought(book)\} \\ \quad goto\_website(Amazon.com) \\ \{\mathsf{B}\,bought(book) \vee \mathsf{G}\,bought(book)\} \end{array}$$

the Conjunction Rule and the Rule for Conditional Actions it is then not difficult to obtain the desired conclusion.

12.9 Conclusion

In this chapter, we discussed a programming logic for GOAL. GOAL and its associated programming logic provide a complete programming theory. The theory includes a concrete proposal for a programming language and a formal, operational semantics for this language as well as a corresponding proof theory based on temporal logic. The logic enables reasoning about the dynamics of agents and about the beliefs and goals of the agent at any particular state during its execution. The semantics of the logic is provided by the GOAL program semantics which guarantees that properties proven in the logic are properties of a GOAL program. In contrast with other attempts (Shoham 1993, Wobcke 2000) to bridge the gap, our programming language and programming logic thus are based on the same semantics. Because of this strong connection between the programming language and its programming logic, we can be sure that statements proven in the logic concern properties of the agent. By providing a formal relation between the agent programming language GOAL and associated agent logic, we were able to bridge the gap between theory and practice. Moreover, a lot of work has already been done in providing practical verification tools for temporal proof theories (Vos 2000), which can be readily used to verify properties of GOAL agents.

There are a number of interesting extensions and problems that still are to be investigated in relation to the programming logic. For example, it would be interesting to develop a semantics for the programming logic for GOAL that would allow the nesting of the belief and goal operators. In the programming logic, we cannot yet nest knowledge modalities which would allow an agent to reason about its own knowledge or that of other agents. Moreover, it is not yet possible to combine the belief and goal modalities. It is therefore not possible for an agent to have a goal to obtain knowledge, nor can an agent have explicit rather than implicit knowledge about its own goals or those of other agents. So far, the use of the B and G operators in GOAL is, first of all, to distinguish between beliefs and goals. Secondly, it enables an agent to express that it does not have a particular belief or goal (consider the difference between $\neg\mathsf{B}\phi$ and $\mathsf{B}\neg\phi$). Another important research issue concerns an extension of the programming framework to incorporate first order languages and extend the programming logic with quantifiers. Finally, more work needs to be done to investigate and classify useful correctness properties of agents. In conclusion, whereas the main aim may be a unified programming framework which includes both declarative goals and planning features, there is still a lot of work to be done to explore and manage the complexities of the language GOAL itself.

CHAPTER 13

Conclusion

The study of intelligent agents has been approached in this thesis by studying agent programming languages. It was argued that agent languages offer a particularly suitable framework for studying intelligent agents because they define in a concise and elegant way the basic concepts associated with agents and provide an implementation platform at the same time. Moreover, a formal approach to agent languages as undertaken here may be one of the most promising roads to bridging the gap between agent theory and the practical engineering of agents.

Part I

In part I, a concrete proposal for an agent language is introduced. The core agent language 3APL is both informally and formally introduced in chapters 2 and 3. The basic concepts are identified and the notions of a *belief*, a *goal*, and *plan* are incorporated into the language. Actually, goals and plans are more or less identified in 3APL since a procedural perspective on goals is taken. Goals in 3APL are so-called *goals-to-do* which specify a plan of action. Plans are stored in a plan library in a 3APL agent by means of so-called *plan rules*. However, plan rules are not the only type of rules that are available to a 3APL agent. A more general type of rule called *practical reasoning rules* provide the agent with reflective and reactive capabilities.

In the agent programming language 3APL, three conceptual levels can be distinguished. That of a belief, a goal, and a practical reasoning rule. These conceptual levels are separated in such a way that only goals are allowed to modify beliefs, and only practical reasoning rules are allowed to modify the goals or plans of an agent. At the most basic level, the beliefs of an agent represent the current situation from the agent's point of view. At the second level, the execution of goals result in updates on the belief base of an agent by adding and deleting information. At the third level, practical reasoning rules supply the agent with reflective capabilities to modify its goals.

In chapter 3, an operational semantics for 3APL is defined. The SOS semantics of (Plotkin 1981) has been used to formally define the meaning of the constructs in the language. From this formal specification, it becomes clear that 3APL can be viewed as a combination of imperative and logic programming. Whereas imperative programming constructs are used to program the usual flow of control from imperative programming and to update the current beliefs of the agent by executing basic actions, logic programming implements the querying of the belief base of the agent and the parameter mechanism of the language based on computing bindings for variables.

Practical reasoning rules are a distinguishing feature of 3APL. In chapter 4, the application and use of these rules is studied in more detail. The practical reasoning rules of 3APL agents provide a powerful mechanism for goal and plan revision. A classification of practical reasoning rules in different rule classes is proposed as a guide to their use. Four classes are identified: failure rules, plan rules, condition-action rules and optimisation rules. The expressive power of rules is illustrated by an application of two types of rules. Condition-action rules are used to implement a post operator for the creation of new goals. Failure rules are used to implement different monitoring facilities. Two such facilities are implemented: disruptors and interrupts. It turned out that the combination of practical reasoning rules and priorities is especially interesting and deserves further study. Priorities provide a means to increase the control over the application of rules.

In chapter 5, the single agent language 3APL introduced in the previous chapters is extended to a multi-agent language. This extension involves the incorporation of communication mechanisms into the language. To this end, two new pairs of primitives are introduced. The first pair is designed in particular for the exchange of information, whereas the second set is designed for the communication of requests. A formal semantics for these primitives has been provided. This semantics aims at capturing what we called the successful processing of a received message by the hearer. The focus in the semantics thus is on the receiver of a message. Moreover, the semantics of communication is designed in such a way that the communication actions fit in naturally with the other constructs of the programming language. In this respect, our approach differs from other approaches that aim at designing a 'universal' communication language for agents like KQML and FIPA-ACL but that do not explain how the communication language can be integrated within an agent framework.

Finally, in chapter 6, the use of the multi-agent language 3APL is illustrated by a meeting scheduling example. A multi-stage negotiation protocol has been used to solve the meeting scheduling problem. The implementation is both natural and concise and thus provides support for the conclusion that the agent programming language offers a powerful and promising alternative to engineering software. In particular with respect to so-called personal assistants, the agent programming paradigm may be preferred over other alternatives.

Part II

In part II, the main theme is a comparison of 3APL with other agent languages. The emphasis is put on a *formal* comparison with languages that are similar enough to allow for such a formal approach. The agent languages that we study are AGENT0, AgentSpeak(L) and ConGolog. These agent languages each attempt to incorporate the most important agent concepts and transform the metaphor of an intelligent agent into an operational framework. Since each of these languages is conceived of in much the same way as 3APL, it is especially appropriate and interesting to compare them. Such a comparison both provides more insight into the basic concepts associated with agents as well as that it provides for an assessment of the unique features within each of the alternative agent languages.

Unfortunately, one of the first agent programming languages proposed in the literature - AGENT0 - does not come with a formal semantics. Since such a formal semantics is a condition for a formal comparison, it is not possible to relate AGENT0 and 3APL in a rigorous and formal way. However, in chapter 7 we take the informal presentation of AGENT0 as a starting point for formalisation and attempt a formal definition of the operational semantics of AGENT0. To this end, we abstracted from a number of features of AGENT0 to capture the core of AGENT0. We called this core the *single agent core of AGENT0.* It includes the beliefs, capabilities, commitments, and commitment rules. Time and communication are not included in the single agent core. For this subset of AGENT0, we then construct an operational semantics.

The design of a formal semantics for AGENT0 reveals the basic differences and similarities between AGENT0 and 3APL. The basic features of AGENT0 are very similar to those of 3APL. The main difference, however, stems from the different types of rules used in both languages. AGENT0 allows the introduction of a new commitment to an action in case a specific commitment is *absent.* 3APL does not allow this, but instead allows the arbitrary revision of goals.

The other two languages that we study in part II, AgentSpeak(L) and ConGolog, are both provided with a formal operational semantics. An operational semantics that formally defines the behaviour of an agent allows for a formal comparison and in chapter 8 we develop a methodology for such a comparison of agent languages. In particular, the method that is developed supports a comparison of the relative *expressive power* of two languages. The method is based on the comparison of the (observable) behaviour of agents. The most important concept developed in this chapter is that of a *translation bisimulation.* A translation bisimulation systematically associates agents from a so-called target language with every agent from a source language. Agents from the source language then can be simulated by agents from the target language if a 'natural' translation exists and the agents of the target language generate the same behaviour as that of the agents from the source language. The conditions under which a translation is considered natural and agents generate the same behaviour are captured in the formal definition of a translation bisimulation.

By using the methodology for comparison of chapter 8, in chapter 9 we show that 3APL has at least the same expressive power of AgentSpeak(L). The proof of this claim focuses on the main conceptual differences between both languages: the concept of an *event* and of an *intention* that are part of AgentSpeak(L) but not of 3APL. It is shown that the former can be eliminated from AgentSpeak(L) without reducing its expressive power and that the latter can be translated into 3APL goals. We conjecture that 3APL has more expressive power than AgentSpeak(L) since only plan rules are needed to simulate AgentSpeak(L). Practical reasoning rules with a more complex structure than the plan rules of AgentSpeak(L) are not used to establish the simulation result.

In the final chapter of part II, the languages ConGolog and 3APL are formally compared. ConGolog is a programming language that extends so-called basic action theories which are specified in the situation calculus with constructs for building composed programs. We show that - given a number of reasonable assumptions - ConGolog can be embedded in 3APL. 3APL thus has at least the expressive power of ConGolog. An interesting conclusion that follows from this result is that basic action theories of the situation calculus can be used to specify the update semantics of basic actions for 3APL agents.

Although the embedding result shows that ConGolog and 3APL are closely related programming languages, the philosophy that inspired the design of both languages is quite different. ConGolog is presented as a high-level programming alternative to planning. The main focus is on extracting a legal action sequence - or a plan - from a nondeterministic program. 3APL is presented as an agent language. The focus is on computing with high-level information represented by the belief base of an agent through the execution of the goals and plans of that agent.

Part III

The design of a programming framework for intelligent agents resulted in part I of this thesis in the programming language 3APL. 3APL supports the construction of intelligent agents, and reflects in a natural way the intentional concepts used to design agents. 3APL is a very powerful agent language, as is shown in part II. It allows the construction of a multi-agent system in which agents communicate their beliefs, requests, perform actions, and construct plans to achieve their goals.

The first two parts thus introduce 3APL and illustrate its power by way of a meeting scheduling example and by means of comparison with other languages. In these parts, however, we do not discuss what an agent logic for 3APL might look like. Since one of the aims of this thesis is to bridge the gap between agent theory and agent practice, in part III we study how such a relation can be established. The starting point that we take is that an agent theory should be more or less similar to a BDI logic (Rao 1996*b*). By making this choice, however, we run into immediate problems. One of the most prominent differences between agent logics like BDI logic and agent programming frameworks like 3APL concerns the notion of a goal. The concept of a goal (or intention) in such logics is a declarative notion, whereas the concept of a goal in most agent programming languages is a procedural notion. As further support for the conclusion that there exists a mismatch between the two, in (Hindriks et al. n.d.) we show that a dynamic logic extended with a belief modality provides an appropriate proof theory for 3APL agents. A modality that corresponds to a motivational attitude of 3APL agents is not present and neither is it very clear how to incorporate such a notion in a proof theory for 3APL.

We conclude that a new road has to be taken to establish the desired link between agent theory and agent programming. To this end, we introduce the agent language GOAL (for Goal Oriented Agent Language) in chapter 11. The main difference between GOAL and 3APL is that GOAL does incorporate a declarative concept of a goal, whereas 3APL does not. A formal, operational semantics for GOAL is defined that can also be used for the design of a a BDI-like agent logic.

In chapter 12, a programming logic for GOAL based on the operational semantics of GOAL is discussed. The GOAL programming logic is a BDI-like logic that enables reasoning about the dynamics of agents and about the beliefs and goals of such agents at any particular state during its execution. GOAL and its associated programming logic provide a complete agent programming theory. The theory includes a concrete proposal for a programming language and a formal, operational semantics for this language as well as a corresponding proof theory based on temporal logic. The GOAL framework thus provides for a concrete proposal to bridge the gap between agent theory and practice.

Future Work

Throughout this thesis the reader will encounter many points of departure for future work. Here, we outline what we think are the more interesting opportunities for future research. As far as the agent language 3APL is concerned, it shares many features with other languages. The two features that make it unique are its notion of a practical reasoning rule and its particular set of communication actions. Although a detailed study of practical reasoning rules was attempted in chapter 4, a lot of work remains to be done. A particularly interesting issue concerns a more detailed study of the interaction between priorities associated with computation steps and the application of rules. Moreover, since practical reasoning rules actually allow the self-modification of an agent program, many issues need to be dealt with. In particular, it becomes much harder to reason about such agent programs and a general agent theory of self-modifying 3APL agents that takes this feature into account needs to be developed. The approach taken in (Hindriks et al. n.d.) does not work in this more general case, and we suggest that a continuation semantics (cf. Nielson & Nielson (1992)) might be used for this purpose.

The formal definition of a semantics for communication only provides a starting point for a much broader investigation that involves the study of communication protocols and the introduction of (sub)group structures into multi-agent systems. As we discussed in chapter 5, even the particular semantics for the communication actions leaves room for many choices. By experimenting with different types of semantics, it might be decided which is to be preferred. As in the case of practical reasoning rules, a proof theory for a multi-agent system of communicating agents still needs to be formulated.

A number of agent languages have been compared in this thesis and it was established that AGENT0, AgentSpeak(L), ConGolog and 3APL form a close family of related agent languages. Despite their similarity, still each of these languages has its own features and it would be interesting to explore in more detail if, and how, features from one framework can be imported within another framework. Since these languages are so closely related to each other, ideas used to create a proof theory, for example, for one of these language could also work for some of the others.

The second agent language that we studied, the language GOAL, we feel, almost raises as many issues as that it answers. GOAL is one of the first concrete proposals to bridge the gap between agent theory and practice, but as such leaves many opportunities for future research. First of all, it is natural to ask why *two* agent languages were introduced in this thesis. It is clear that both languages include different features, but why did we not combine these features into a single language? The problem is that this is quite hard. To integrate the more sophisticated planning capabilities of 3APL into GOAL would almost certainly lead us back to the question what a proof theory for such a combination would look like. To integrate declarative goals into 3APL, however, also raises many new issues. For example, how do we manage the interaction between a declarative goal base and a dynamic plan base (the procedural goal base of 3APL). At the moment, we do not know how to solve these problems. A more promising extension of GOAL involves the incorporation of communication into the language. Because GOAL includes declarative goals, it may be a realistic option and indeed would be very interesting to see whether or not a semantics for communication that is more similar to that of speech acts can be developed for GOAL.

Apart from the programming language GOAL, the programming logic for GOAL raises just as many interesting issues. For example, it would be very interesting to develop a semantics for the programming logic for GOAL that would allow the nesting of the belief and goal operators. This is not yet possible and an agent cannot yet reason about its own knowledge or that of other agents. Neither is it possible to combine the belief and goal modalities. Another important research issue concerns an extension of the programming framework to incorporate first order languages and extend the programming logic to a first order logic. Finally, more work needs to be done to investigate and classify useful correctness properties of agents.

We conclude with a general remark. Our aim has been to show that intelligent agents provide a very promising new programming paradigm and to support this claim by providing a concrete framework for programming agents. Even if the reader is not convinced by our work, our hope is that we convincingly showed the use of formal techniques and ideas from the area of programming semantics and concurrency theory as a means to clarify the notion of an intelligent agent that was the major theme of this thesis. We believe it remains fruitful to explore and exploit ideas and techniques from these areas as much as possible.

BIBLIOGRAPHY

Andrews, Gregory R. (1991), *Concurrent Programming: Principles and Practice*, The Benjamin/Cummings Publishing Company.

Bach, Kent & Robert M. Harnish (1979), *Linguistic Communication and Speech Acts*, MIT Press.

Baecker, Ronald M. (1993), *Readings in Groupware and Computer Supported Cooperative Work: Software to Facilitate Human-Human Collaboration*, Morgan Kaufman.

Bratman, Michael E. (1987), *Intentions, Plans, and Practical Reasoning*, Harvard University Press.

Chandy, K. Mani & Jayadev Misra (1988), *Parallel Program Design*, Addison-Wesley.

Chellas, Brian F. (1980), *Modal logic: an introduction*, Cambridge University Press.

Cohen, Philip R. & Hector J. Levesque (1990*a*), 'Intention is choice with commitment', *Artificial Intelligence* **42**, 213–261.

Cohen, Philip R. & Hector J. Levesque (1990*b*), Rational Interaction as the Basis for Communication, *in* P.Cohen, J.Morgan & M.Pollack, eds, 'Intentions in Communication', MIT Press.

Cohen, Philp R. & Hector J. Levesque (1995), Communicative Actions for Artificial Agents, *in* 'Proceedings of the International Conference on Multi-Agent Systems', AAAI Press.

Colombetti, Marco (2000), Semantic, Normative and Practical Aspects of Agent Communication, *in* F.Dignum & M.Greaves, eds, 'Issues in Agent Communication', Springer-Verlag, pp. 17–30.

Conry, Susan E., Robert A. Meyer & Victor R. Lesser (1988), Multistage Negotiation in Distributed Planning, *in* A.Bond & L.Gasser, eds, 'Readings in Distributed Artificial Intelligence', Morgan Kaufman, pp. 367–384.

d'Inverno, Mark, David Kinny, Michael Luck & Michael Wooldridge (1998), A Formal Specification of dMARS, *in* M.Singh, A.Rao & M.Wooldridge, eds, 'Intelligent Agents IV (LNAI 1365)', pp. 155–176.

d'Inverno, Mark & Michael Luck (1998), 'Engineering AgentSpeak(L): A Formal Computational Model', *Journal of Logic and Computation* **8**(3).

Dragoni, Aldo F., Paolo Giorgini & Luciano Serafini (to appear), Updating Mental States from Communication, *in* C.Castelfranchi & Y.Lésperance, eds, 'Intelligent Agents VII', Springer-Verlag.

Dunin-Keplicz, Barbara & Jan Treur (1995), Compositional Formal Specification of Multi-Agent Systems, *in* M.Wooldridge & N.Jennings, eds, 'Intelligent Agents (LNAI 890)', Springer-Verlag, pp. 102–117.

Eijk, Rogier M. van, Frank S. de Boer, Wiebe van der Hoek & John-Jules Ch. Meyer (1998), Systems of communicating agents, *in* H.Prade, ed., 'Proceedings of 13th biennial European Conference on Artificial Intelligence (ECAI'98)', John Wiley and Sons, pp. 293–297.

Eijk, Rogier M. van, Frank S. de Boer, Wiebe van der Hoek & John-Jules Ch. Meyer (1999), Information-Passing and Belief Revision in Multi-Agent Systems, *in* J. P. M.Müller, M. P.Singh & A. S.Rao, eds, 'Intelligent Agents V (LNAI 1555)', Springer-Verlag, pp. 29–45.

Enderton, Herbert B. (1972), *A Mathematical Introduction to Logic*, Academic Press.

Felleisen, Matthias (1990), On the expressive power of programming languages, *in* G.Goos & J.Hartmanis, eds, '3rd European Symposium on Programming (LNCS 432)', Springer-Verlag, pp. 134–151.

FIPA (1998), 'FIPA 97 Specification, Part 2, Agent Communication Language'.

Fisher, Michael (1994), A Survey of Concurrent MetateM - The Language and its Applications, *in* 'Proceedings of First International Conference on Temporal Logic (LNCS 827)', Springer-Verlag, pp. 480–505.

Franklin, Stanley P. & Arthur C. Graesser (1997), Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, *in* 'Intelligent Agents III (LNAI 1193)', Springer-Verlag, pp. 21–36.

Gärdenfors, Peter (1988), *Knowledge in Flux: Modelling the Dynamics of Epistemic States*, MIT Press.

Giacomo, Giuseppe de, Yves Lespérance & Hector Levesque (2000), '*ConGolog*, a Concurrent Programming Language Based on the Situation Calculus', *Artificial Intelligence* **121**(1-2), 109–169.

Giacomo, Guiseppe de & Hector J. Levesque (1998), An incremental interpreter for high-level programs with sensing, Technical report, Department of Computer Science, University of Toronto.

Giacomo, Guiseppe de, Yves Lespérance & Hector J. Levesque (1997), Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus, *in* M. E.Pollack, ed., 'Proceedings of the fifteenth International Joint Conference on Artificial Intelligence', Morgan Kaufman, pp. 1221–1226.

Groenendijk, Jeroen & Martin Stokhof (1984), On the Semantics of Questions and the Pragmatics of Answers, *in* F.Landman & F.Veltman, eds, 'Varieties of Formal Semantics', Foris, pp. 143–170.

Groote, Jan Friso (1993), 'Transition System Specifications with Negative Premises', *Theoretical Computer Science* **118**(2), 263–299.

Harel, David (1979), *First-order dynamic logic (LNCS 68)*, Springer-Verlag.

Hindriks, Koen V., Frank S. de Boer, Wiebe van der Hoek & John-Jules Ch. Meyer (1998), Formal Semantics for an Abstract Agent Programming Language, *in* M.Singh, A.Rao & M.Wooldridge, eds, 'Intelligent Agents IV (LNAI 1365)', Springer-Verlag, pp. 215–229.

Hindriks, Koen V., Frank S. de Boer, Wiebe van der Hoek & John-Jules Ch. Meyer (1999*a*), 'Agent Programming in 3APL', *Autonomous Agents and Multi-Agent Systems* **2**(4), 357–401.

Hindriks, Koen V., Frank S. de Boer, Wiebe van der Hoek & John-Jules Ch. Meyer (1999*b*), Control Structures of Rule-Based Agent Languages, *in* J. P.Müller, M. P.Singh & A. S.Rao, eds, 'Intelligent Agents V (LNAI 1555)', Springer-Verlag, pp. 381–396.

Hindriks, Koen V., Frank S. de Boer, Wiebe van der Hoek & John-Jules Ch. Meyer (n.d.), 'A Programming Logic for the Agent Programming Language 3APL', *to appear* .

Hindriks, Koen V., Frank S. de Boer, Wiebe van der Hoek & John-Jules Meyer (1999*c*), An Operational Semantics for the Single Agent Core of AGENT-0, Technical Report UU-CS-1999-30, Department of Computer Science, University Utrecht.

Hoare, C.A.R. (1985), *Communicating Sequential Processes*, Prentice Hall.

Hoek, Wiebe van der, Bernd van Linder & John-Jules Ch. Meyer (1994), A logic of capabilities, *in* A.Nerode & Y.Matiyasevich, eds, 'Proc. of the third int. symposium on the logical foundations of computer science (LNCS 813)', Springer-Verlag, pp. 366–378.

Hoyle, Michelle A. & Christopher Lueg (1997), Open Sesame!: A Look at Personal Assistants, *in* 'Proceedings of the International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM 97)', pp. 51–60.

Jennings, Nick R. & A.J. Jackson (1995), 'Agent based meeting scheduling: A Design and Implementation', *Electronics Letters, The Institution of Electrical Engineering* **31**(5), 350–352.

Jones, Andrew J.I. (1993), 'Practical Reasoning, California-style: Some Remarks on Shoham's Agent-oriented Programming (AOP)'.

Kautz, Henry A., Bart Selman, Michael Coen, Steven Ketchpel & Chris Ramming (1994), An Experiment in the Design of Software Agents, *in* 'Proc. of AAAI-94', AAAI Press.

Labrou, Yannis & Tim Finin (1994), A semantics approach for KQML - a general purpose communication language for software agents, *in* 'Third International Conference on Information and Knowledge Management (CIKM'94)'.

Lennon, Jennifer & Arnould Vermeer (1995), 'From Personal Computer to Personal Assistant', *Journal of Universal Computer Science* **1**(6), 410–422.

Lespérance, Yves, Hector J. Levesque, Fanghzen Lin, Daniel Marcu, Ray Reiter & Richard B. Scherl (1996), Foundations of a Logical Approach to Agent Programming, *in* M.Wooldridge, J.Müller & M.Tambe, eds, 'Intelligent Agents II (LNAI 1037)', Springer-Verlag, pp. 331–346.

Levesque, Hector J., Ray Reiter, Yves Lespérance, Fangzhen Lin & Richard B. Scherl (1997), 'GOLOG: A logic programming language for dynamic domains', *Journal of Logic Programming* **31**, 59–84.

Lin, Fangzhen & Ray Reiter (1997), 'How to Progress a Database', *Artificial Intelligence* **92**, 131–167.

Linder, Bernd van, Wiebe van der Hoek & John-Jules Ch. Meyer (1996), Formalising motivational attitudes of agents: On preferences, goals, and commitments, *in* M.Wooldridge, J.Müller & M.Tambe, eds, 'Intelligent agents II (LNAI 1037)', Springer-Verlag, pp. 17–32.

Lloyd, John W. (1987), *Foundations of Logic Programming*, Springer-Verlag.

Maes, Pattie (1994), 'Agents that Reduce Work and Information Overload', *Communications of the ACM* **37**(7), 30–40.

Manna, Zohar & Amir Pnueli (1992), *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag.

Mayer, Martha C. & Fiora Pirri (1996), 'Abduction is not Deduction-in-Reverse', *Journal of the Interest Group in Pure and Applied Logics* **4**(1), 1–14.

McCarthy, John & Patrick J. Hayes (1969), Some philosophical problems from the standpoint of artificial intelligence, *in* Meltzer & Michie, eds, 'Machine Intelligence', Edinburgh University Press, pp. 463–502.

Mendelson, Elliott (1979), *Introduction to Mathematical Logic*, D. Van Nostrand Company.

Meyer, John-Jules Ch., Wiebe van der Hoek & Bernd van Linder (1999), 'A Logical Approach to the Dynamics of Commitments', *Aritificial Intelligence* **113**, 1–40.

Milner, Robin (1989), *Communication and Concurrency*, Prentice Hall.

Minker, Jack, ed. (1988), *Foundations of deductive databases and logic programming*, Morgan Kaufmann.

Nielson, Hanne Riis & Flemming Nielson (1992), *Semantics with Applications: A Formal Introduction*, Wiley.

Oérez, Asunción Gómez & V. Richard Benjamins (1999), Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods, *in* V.Benjamins, ed., 'Proceedings of the IJCAI-99 workshop on Ontologies and Problem-Solving Methods (KRR5)', CEUR Publications (http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/).

Papadopoulos, George A. & Farhad Arbab (1998), Coordination models and languages, *in* 'Advances in Computers: The Engineering of Large Systems', Academic Press, pp. 329–400.

Park, David M. R. (1980), *Concurrency and Automata on Infinite Sequences (LNCS 104)*, Springer-Verlag.

Pirri, Fiora & Ray Reiter (1999), 'Some Contributions to the Metatheory of the Situation Calculus', *Journal of the ACM* **46**(3), 261–325.

Plotkin, G. (1981), A structural approach to operational semantics, Technical report, Aarhus University, Computer Science Department.

Poggi, Agostini (1995), DAISY: An object-oriented system for distributed artificial intelligence, *in* M.Wooldridge & N.Jennings, eds, 'Intelligent Agents (LNAI 890)', Springer-Verlag, pp. 341–354.

Poole, David (1989), 'Explanation and prediction: An architecture for default and abductive reasoning', *Computational Intelligence* **5**(1).

Rao, Anand S. (1996*a*), AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, *in* W.van der Velde & J.Perram, eds, 'Agents Breaking Away (LNAI 1038)', Springer-Verlag, pp. 42–55.

Rao, Anand S. (1996*b*), Decision Procedures for Propositional Linear-Time Belief-Desire-Intention Logics, *in* M.Wooldridge, J.Müller & M.Tambe, eds, 'Intelligent Agents II (LNAI 1037)', Springer-Verlag, pp. 33–48.

Rao, Anand S. (1997), 'Private communication'.

Rao, Anand S. & Michael P. Georgeff (1990), Intentions and Rational Commitment, Technical Report 8, Australian Artificial Intelligence Institute, Melbourne, Australia.

Reiter, Ray (1991), The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression, *in* 'Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy', Academic Press, pp. 359–380.

Searle, John R. (1969), *Speech acts*, Cambridge University Press.

Segerberg, Krister (1970), 'Modal logics with linear alternative relations', *Theoria* **36**, 301–322.

Sen, Sandip & Edmund Durfee (1996), 'A Contracting Model for Flexible Distributed Scheduling', *Annals of Operations Research* **65**, 195–222.

Shoham, Yoav (1991), Implementing the Intentional Stance, *in* R.Cummins & J.Pollock, eds, 'Philosophy and AI: Essays at the Interface', MIT Press, chapter 11, pp. 261–277.

Shoham, Yoav (1993), 'Agent-oriented programming', *Artificial Intelligence* **60**, 51–92.

Shoham, Yoav (1994), Agent Oriented Programming: An overview of the framework and summary of recent research, *in* J.-M. F.Masuch & L.Pólos, eds, 'Knowledge representation and reasoning under uncertainty, logic at work (LNAI 808)', Springer-Verlag, pp. 123–129.

Singh, Munindar .P. (1994), *Multiagent systems (LNAI 799)*, Springer-Verlag.

Stefik, Mark (1995), *Introduction to Knowledge Systems*, Morgan Kaufman.

Tennent, Robert D. (1991), *Semantics of Programming Languages*, Prentice Hall.

Thomas, Sarah Rebecca (1993), PLACA, An Agent Oriented Programming Language, PhD thesis, Department of Computer Science, Stanford University.

Torrance, Mark C. (1991), 'The AGENT0 Manual'.

Vos, Tanja (2000), UNITY in Diversity, PhD thesis, Department of Computer Science, Utrecht University.

Wobcke, Wayne (2000), On the Correctness of PRS Agent Programs, *in* N.R. Jennings and Y. Lespérance, ed., 'Intelligent Agents VI (LNAI 1757)', Springer-Verlag.

Wooldridge, Michael J. (1997), A Knowledge-Theoretic Semantics for Concurrent MetateM, *in* J.Müller, M.Wooldridge & N.Jennings, eds, 'Intelligent Agents III (LNAI 1193)', Springer, pp. 357–374.

Samenvatting

In dit proefschrift wordt een nieuwe visie (*paradigma*) ontwikkeld op hoe een software probleem zou kunnen worden aangepakt. Een paradigma biedt handvaten voor het ontwikkelen van software en probeert een brug te slaan tussen de structuur van het alledaagse denken en de manier waarop software opereert. Hier wordt er dus naar gestreefd om de visie op het ontwikkelen van software zo natuurlijk mogelijk aan te laten sluiten bij de structuur van denken van het alledaagse denken.

Het idee is om een programma te zien als een assistent (een *intelligent agent*) die het initiatief neemt om de gebruiker te helpen bij het uitvoeren van bepaalde activiteiten. Een intelligent agent kan opdrachten ontvangen in een taal die zowel voor de agent als voor de gebruiker van de agent goed te begrijpen is.

Het belangrijkste probleem bestaat uit het vinden van een dergelijke taal. Omdat deze taal moet aansluiten bij het alledaagse denken, wordt begonnen met het analyseren van alledaagse begrippen. In de taal van alledag spelen drie begrippen een belangrijke rol: *kennis*, *doelen*, en *plannen*. Kennis verwijst naar alles wat we weten. Een doel verwijst naar wat we willen bereiken. En een plan beschrijft op welke manier een doel kan worden bereikt. Verder moet men om een plan uit te kunnen voeren over de juiste vaardigheden beschikken. *Vaardigheid* is het vierde en laatste begrip dat wordt gebruikt om een intelligent agent te definiëren.

De vier begrippen die werden gedestilleerd uit de alledaagse taal leveren dus de bouwstenen voor een nieuw agent paradigma. Om intelligent agents te ontwikkelen, moet een een programma worden uitgerust met de juiste vaardigheden, kennis, doelen en plannen. In het bijzonder zijn we geïnteresseerd hoe een intelligent agent kan worden geïnstrueerd oftewel *geprogrammeerd.*

Programmeertalen voor intelligent agents is het hoofdthema van dit proefschrift. In het eerste deel worden de vier basisbegrippen die hierboven werden opgesomd gebruikt voor het ontwerpen van de agent programmeertaal 3APL.

In hoofdstuk 2 wordt 3APL geïntroduceerd. Een programma in 3APL is een intelligent agent die in staat is bepaalde acties uit te voeren, kan worden uitgerust met initiële kennis en doelen, en plannen kan construeren om die doelen te bereiken. Plannen kunnen worden bedacht door een agent met behulp van bepaalde *regels*. In 3APL worden zulke regels *practical reasoning rules* genoemd. Een regel vertelt een agent welk plan bij een bepaald doel en bepaalde situatie past. Als ik bijvoorbeeld de eerstvolgende trein naar Groningen wil halen en weet dat de trein om 20 over 4 vertrekt, dan is het plan om om 10 voor 4 de bus naar het station te pakken vanaf de Uithof geschikt.

Het is belangrijk om de betekenis van elk van de vier agent componenten duidelijk uiteen te zetten. Dat is om verschillende redenen belangrijk. Eén reden is dat een programmeur om te kunnen programmeren in de taal moet begrijpen hoe een 3APL agent werkt. Een andere reden is dat het precies vastleggen van de betekenis van de taal ook duidelijk maakt wat er allemaal wel en niet mogelijk is met de taal. Een wiskundige definitie van de betekenis legt de uitdrukkingskracht of expressiviteit van de taal ondubbelzinnig vast.

Een formele definitie van de taal 3APL wordt in hoofdstuk 3 gegeven met behulp van een *operationele semantiek*. Een operationele semantiek beschrijft welke acties een agent in een bepaalde toestand kan uitvoeren. Uit deze formele beschrijving wordt duidelijk dat de focus in 3APL ligt op de *mentale wereld* van een agent. Een agent kan door middel van acties zijn kennistoestand veranderen en plannen bedenken om een doel te bereiken. Een concreet verband met de omgeving waarin de agent 'leeft' kan echter niet worden vastgelegd met 3APL. Het is dus wel mogelijk om een robot te programmeren met 3APL, maar de fysieke aansturing van de robot's camera's en wielen moet als een apart software probleem gezien worden. Het voordeel hiervan is dat bij het programmeren van een robot de verschillende deelproblemen goed uit elkaar kunnen worden gehouden.

Een bijzondere eigenschap van 3APL agents is dat ze niet alleen plannen kunnen bedenken, maar ze op een later tijdstip eventueel ook weer kunnen wijzigen als dat beter uitkomt. Ook al mag dit de lezer als vanzelfsprekend voorkomen, binnen de informatica is dit een controversieel onderwerp. Een 3APL agent is namelijk een programma dat zichzelf kan modificeren en dit type programma's zijn een stuk complexer dan programma's die zichzelf niet modificeren. Reden genoeg om een heel hoofdstuk te wijden aan de *practical reasoning rules* die deze zelfmodificatie mogelijk maken.

In de hoofstukken 5 en 6 wordt 3APL vervolgens uitgebouwd tot een multi-agent taal en wordt een applicatie voor het plannen van tijdstippen voor vergaderingen beschreven. In multi-agent 3APL kunnen meerdere agents met elkaar communiceren om informatie uit te wisselen en elkaar om hulp te vragen. In de applicatie wordt communicatie tussen meerdere agents gebruikt om te onderhandelen over een passend tijdstip om elkaar te ontmoeten. Hierbij moet in het oog worden gehouden dat de agents voor hun gebruikers onderhandelen, maar de gebruiker van een agent zelf zich aan het door de agent afgesproken tijdstip moet houden.

In het tweede deel wordt 3APL vergeleken met een drietal andere agent programmeertalen: AGENT0, AgentSpeak(L) en ConGolog. AGENT0 is een programmeertaal die niet formeel gedefinieerd maar informeel geïntroduceerd is. Omdat toch vooral een formele beschrijving precies maakt wat de verschillen en overeenkomsten tussen talen zijn, wordt een operationele semantiek ontworpen voor een substantieel fragment van AGENT0 in hoofdstuk 7. In hoofdstuk 8 komen we terug op de expressiviteit van 3APL die formeel is vastgelegd door de semantiek van de taal. In dit hoofdstuk wordt een methode bestudeerd voor het vergelijken van de uitdrukkingskracht van verschillende (agent) programmeertalen. Deze methode wordt in hoofdstuk 9 en 10 gebruikt om formeel te laten zien dat 3APL dezelfde uitdrukkingskracht heeft als AgentSpeak(L) en ConGolog en misschien meer.

In het derde deel bespreken we een belangrijk probleem uit het agent onderzoek. Tot zover heeft het onderzoek in dit proefschrift zich voornamelijk gericht op programmeertalen voor agents. Een programmeertaal maakt het mogelijk om vast te leggen (te programmeren) *hoe* een agent iets doet. Een agent programma legt meer de nadruk op de operationele werking van een agent. Het is ook mogelijk meer de nadruk te leggen op de specificatie van *wat* een agent doet. Specificaties worden opgeschreven in een *logische taal*. Tussen programmeertalen en logische talen bestond tot nu toe echter nog altijd een kloof in het agentonderzoek. In hoofdstuk 11 wordt de oorzaak voor deze kloof geïdentificeerd. Het probleem is dat een doel in een programmeertaal een operationele betekenis heeft, terwijl in logische talen een doel een omschrijving is van een gewenste toestand. In een programmeertaal lijkt een doel op een plan dat bijvoorbeeld beschrijft dat je je jas aan moet trekken, en daarna de bus moet pakken. In een logische taal beschrijft een doel de eindtoestand dat je de trein wilt halen. Om dit verschil te kunnen overbruggen wordt een nieuwe programmeertaal GOAL gedefinieerd waarin een doel beschrijft wat er moet worden bereikt. Voor deze programmeertaal is het mogelijk een temporele logica te ontwerpen die gebruikt kan worden als specificatietaal voor GOAL agents.

De taal GOAL overbrugt de kloof tussen agent logica's en agent programmeertalen, maar bevat helaas niet alle planning mogelijkheden van 3APL. Het is een interessante open vraag voor vervolgonderzoek om te bepalen of het mogelijk is het beste van beide talen te combineren.

Curriculum Vitae

Koen Hindriks werd op 29 december 1971 geboren te Groningen. In augustus 1990 behaalde hij het VWO diploma aan het Gomarus College eveneens in Groningen. In september van datzelfde jaar begon hij met de studie Informatica aan de Rijksuniversiteit Groningen. Hij volgde de specialisatierichting Theoretische Informatica en studeerde in 1996 af op het onderwerp Logica's voor Multi-Agent Systemen.

Vanaf half oktober 1996 tot en met oktober 2000 was hij in dienst van de Universiteit Utrecht als AIO in de groep Intelligent Systems van Prof. dr. John-Jules Ch. Meyer. Hij verrichtte daar het in dit proefschrift beschreven onderzoek. Gedurende dit onderzoek werkte hij in 1999 vanaf april tot en met juni als gastonderzoeker van Prof. dr. Hector Levesque aan de University of Toronto. In november 2000 trad hij in dienst bij Andersen Consulting te Amsterdam.