COGNITIVE AGENT PROGRAMMING

A SEMANTIC APPROACH

SIKS Dissertation Series No. 2006-19

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

Printed by Gildeprint drukkerijen B.V., Enschede

ISBN-10: 90-393-4355-1
ISBN-13: 978-90-393-4355-5

Cognitive Agent Programming

A Semantic Approach

Programmeren van Cognitieve Agenten

Een Semantische Benadering

(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. W.H. Gispen,
ingevolge het besluit van het college voor promoties
in het openbaar te verdedigen
op woensdag 25 oktober 2006 des middags te 12.45 uur door

Maria Birna van Riemsdijk

geboren op 29 oktober 1978, te Wageningen

promotoren:	Prof. Dr. John-Jules Ch. Meyer
co-promotoren:	Dr. Frank S. de Boer
	Dr. Mehdi Dastani

Het hoogste doel in het leven is de groei van het ik, de vergroting van het bewustzijn. Het hoogste geluk ligt in de vreugde over deze groei, over het gevoel meer mens te worden.

The ultimate goal in life is the growth of self, the expansion of consciousness. The ultimate happiness lies in the joy about this growth, in the feeling of becoming more human. - *Translated by Targetranslations, Apeldoorn.*

Bertus Mulder, *Gerrit Thomas Rietveld: Leven Denken Werken*

Contents

Preface

Simplicity is my holy grail. Only through a striving for simplicity can we make progress.

Striving for simplicity is important in the sense of identifying an issue and simplifying it such that its essence can be studied in isolation. This is what the famous Dutch computer scientist Dijkstra (1930 - 2002) has referred to as "separation of concerns": "Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. [...] It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of." [**?**]

Contrary to what one might think, this is not a trivial undertaking. One should not *over*simplify matters and ignore essential aspects of the issue. That is, it is not the case that simpler is always better. Rather, I argue that when proposing a certain approach, one should ideally be able to argue for each aspect of the approach why this aspect is incorporated. Through this, one can achieve a clarity of presentation and a true understanding of the subject matter.

Unfortunately, simplicity doesn't always sell, as illustrated by the following quote from Dijkstra: "When you give for an academic audience a lecture that is crystal clear from alpha to omega, your audience feels cheated and leaves the lecture hall commenting to each other: 'That was rather trivial, wasn't it?' The sore truth is that complexity sells better." [**?**]

A solution to this problem is suggested in the following poem, which is attributed to[1] the Danish artist and scientist Piet Hein (1905 - 1996).

If you want the world to hear
Write your papers crystal clear,
then add some ingenuities
to show how hard to do it is...

[1] I found this poem on the website of computer scientist Sophia Drossopoulou: `http://www.doc.ic.ac.uk/~scd/`.

Besides the scientific progress that can be achieved through a striving for simplicity, a simple and clear theory can be enjoyed as a thing of beauty. This is where science and art meet.

> *And this is the work of an artist. The artist, who thanks to his limited talents is able to make some clear distinctions, he enables us to see some elements of fundamental reality, which up until then escaped us.*
>
> *The plastic artist for example distinguishes colour, shape, space and movement as elements of the seeing and restricts himself to one of these four elements only to distinguish more clearly the phenomena of reality which come flashing by. The clarification through this of our look onto life is the perception: beauty. Our world expands itself and we rejoice in its beauty.*[2]
>
> Gerrit Th. Rietveld, *GR 263*

Like looking at art, practicing science allows one to continually discover new viewpoints and expand one's horizon. It is this which makes me enjoy doing research.

Ok, I'll get off my soapbox now and start thanking some people. I thank, first of all, my supervisors John-Jules Ch. Meyer, Frank S. de Boer, and Mehdi Dastani. I thank John-Jules for his open mind and cheerful spirit. I couldn't have wished for a better promotor! John-Jules always took the time to listen to me and to give advise on anything ranging from technical matters to advise on what to do after finishing the thesis. I thank Frank for contributing his great technical knowledge and insight, for many interesting discussions, and for his humor which always made our meetings enjoyable. I thank Mehdi for always making me think about why I chose a certain approach, and for inviting me to contribute to many of his projects, whether it was writing papers or giving tutorials. Mehdi is never short of ideas for doing research, and has been great company on the many conferences that we attended together.

I thank Michael Fisher, Koen Hindriks, Joost Kok, Leon van der Torre, and Jan Treur for agreeing to be a member of the reading committee and taking the time to read my thesis. I thank my colleagues, from the Intelligent Systems group and others, for the many ways in which they have contributed to this thesis. I thank in particular Huib Aldewereld, Rafael Bordini, Lars Braubach,

[2] Translated by Targetranslations, Apeldoorn. Original text: "En dit is het werk van de kunstenaar. De kunstenaar, die dank zij zijn beperking van aanleg, enkele heldere onderscheidingen maakt, laat ons enige stukjes primaire werkelijkheid zien, die ons voorheen ontgingen. De plastische kunstenaar b.v. onderscheidt kleur, vorm, ruimte, en beweging als onderdelen van het zien en bepaalt zich tot een van deze vier, om met des te meer aandacht de voorbij flitsende verschijnselen der werkelijkheid te onderscheiden. De verheldering hierdoor van ons levensbeeld is de gewaarwording: schoonheid. Onze wereld verruimt zich en wij verheugen ons in z'n schoonheid."

Martin Bravenboer, Jan Broersen, Martin Caminada, Jurriaan van Diggelen, Frank and Virginia Dignum, Davide Grossi, Paul Harrenstein, Koen Hindriks, John Horty, Joris Hulstijn, Wiebe van der Hoek, Geert Jonker, Henk-Jan Lebbink, Viviana Mascardi, Peter Novak, Lin Padgham, Cees Pierik, Alexander Pokahr, Henry Prakken, Carsten Riggelsen, Sebastian Sardina, Liz Sonenberg, the PhD students from the SIKS research school, Leon van der Torre, Javier Vázquez-Salceda, Bob van der Vecht, Michael Winikoff, and Cees Witteveen. Extra special thanks go to Paul Harrenstein for letting me use his thesis style file (and for providing the indispensable accompanying technical support), and to Huib Aldewereld for being incredably patient in answering my numerous (usually not so interesting) questions about computers, and for always being willing to help out when I was (again) completely at a loss as to how to make the computer do what I wanted.

I thank Wilke Schram and the Department of Information and Computing Sciences of Utrecht University for giving me the opportunity to travel to many conferences which has taken me to beautiful places such as Melbourne, New York, Estonia, Lisbon, Covilha, Dagstuhl and Scotland. I thank AgentLink and Mehdi Dastani for inviting me to attend Technical Forum Groups in Ljubljana and Budapest, and the AAMAS organization for supporting my attendance of AAMAS'03 (Melbourne), AAMAS'04 (New York), and AAMAS'06 (Japan). Also I am grateful for having received scholarships from the organization of the CLIMA workshop, allowing me to attend CLIMA'05 in London, even though I did not have an accepted paper, and CLIMA'06 in Japan. I thank Richard Starmans and SIKS for many SIKS courses, and for supporting my attendance of DEON'06.

I thank my roommates Huib Aldewereld, Virginia Dignum, Geert Jonker, Cees Pierik, Wieke de Vries, and the other members of the Intelligent Systems group for the pleasant working atmosphere. I have probably learned more about the world through the many discussions over lunch than from watching the evening news, and I will really miss having a "bakkie pleur" in the afternoon with my fellow PhD students. I thank especially Geert for enduring me chatting about being "almost finished" writing the thesis, about filling out forms, about the next version 1.7.x of my thesis cover, etc. I will miss Geert's humor, and conversations about hats, cars, friendship, and the like. I thank Cees Pierik and Paul Harrenstein for being my paranimfen. I will miss Cees' down-to-earth attitude to life, and the pleasant talks about anything from jogging to religion, and I am happy not to have to miss Paul.

I thank my friends and family for being there for me over the years. I thank especially my brother Rombout for always being willing to help, and my parents Trudy and Willem for giving advise on anything from cover design to buying a house, and for always being there to share experiences with.

All of you have made the past four years the enjoyable experience it has been.

Utrecht, August 2006

Chapter 1

Introduction

The work described in this thesis draws mainly on research done in the agent systems field and on research concerning programming languages. That is, in this thesis we are concerned with the design and investigation of dedicated programming languages for programming agents. In Sections 1.1 and 1.2, respectively, we provide some background on those aspects of the agent systems field and of programming language research that are relevant in the context of this thesis.

An agent is commonly seen as "an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives" [Jennings, 1999].[1]

An autonomous computing entity encapsulates its state and makes decisions about what to do based on this state, without the direct intervention of humans or others [Wooldridge, 1997]. Agents are situated in some environment which can change during the execution of the agent. This requires flexible problem solving behavior, which means that the agent should be able to respond adequately to changes in its environment in such a way that it achieves its objectives or goals. Such flexibly behaving computing entities that are able to make "good" decisions about what to do, are often called intelligent agents or *rational* agents [Rao and Georgeff, 1991, van der Hoek et al., 1998, van der Hoek and Wooldridge, 2003].

Domains in which agents are viewed as providing added value, are typically dynamic and complex, such as the domains of business process management [Jennings et al., 1996] and disaster rescue [Visser and Burkhard, 2006].

[1] In [Jennings, 1999], this definition of what an agent is, is attributed to [Wooldridge, 1997]. However, the definition cannot be found literally in the paper by Wooldridge.

1.1 Cognitive Agent Programming

Programming rational agents is not a trivial task. First, one needs to establish what it means that an agent is rational, i.e., one has to develop a *theory* of rational agents. The next step then needs to address how these theoretical ideas of rational agency can be implemented in a concrete piece of software. The latter can, broadly speaking, be approached in two ways. One possibility is to develop an *architecture* for rational agents, specifying what general structure a rational agent should have. Such an architecture can then be implemented using some general purpose programming language. Another possibility is to design a *dedicated programming language* for programming rational agents.

Research into agent theories, architectures, and languages has a strong tradition[2] in the agent systems field (see also [Wooldridge and Jennings, 1995]). The work as presented in this thesis builds on this tradition.

The origins of much research into agent theories, architectures, and languages lie with Bratman's so-called Belief Desire Intention (*BDI*) philosophy [Bratman, 1987] (see also Section 5.1.1). BDI philosophy in turn can be viewed as based on Dennett's intentional stance [Dennett, 1987]. The idea of the intentional stance is that the behavior of rational agents can be predicted by ascribing *beliefs* and *desires* to the agent, and by assuming that the agent will tend to act in pursuit of its desires, taking into account its beliefs about the world.

The idea of Bratman now is that beliefs and desires are not enough for explaining and describing rational behavior. He argues that there is another notion, i.e., *intention*, that is essential for an account of practical rationality. His view is that an agent may have many, possibly conflicting, desires. It is not feasible for an agent to continually weigh all its desires, in order to decide what to do. Therefore, an agent settles at some point on achieving a certain (non-conflicting) subset of its desires. These chosen desires - and corresponding courses of action - are the agent's intentions.

Intentions have a characteristic stability, in the sense that an agent will in principle not reconsider previously formed intentions, except if a significant problem presents itself. Also, if an agent wants to adopt new intentions, these new intentions should not conflict with already existing intentions. Existing intentions thus form a so-called *screen of admissibility*.

BDI philosophy suggests a particular view on rational agency. It is this view that has formed the basis for the development of several *BDI logics* for modeling and reasoning about rational agents [Cohen and Levesque, 1990] [Rao and Georgeff, 1991, van der Hoek et al., 1998, Rao and Georgeff, 1998] (see also Section 5.1.2). The notions of which these logics investigate properties are the notions of beliefs, desires, and intentions, but also related notions

[2]This is demonstrated by a series of successful Agent Theories, Architectures, and Languages (ATAL) workshops which were held annually from 1994 to 2001. Today, ATAL is part of the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS).

such as goals, wishes, opportunity, capability, commitments, etc. Such notions are often referred to as *mental attitudes*.

Shortly after the first BDI logics were proposed, it was Shoham who first put forward the idea to *program* rational agents *using* mental attitudes [Shoham, 1993]. Shoham proposes the agent programming language AGENT-0, and coins the term *agent-oriented programming*. While Shoham proposes a programming language, others have proposed a specific BDI *architecture* for the implementation of rational agents [Georgeff and Lansky, 1987] [Ingrand et al., 1992]. The resulting system was called the Procedural Reasoning System (PRS).

Based on the PRS architecture, the agent programming language AgentSpeak(L) was proposed [Rao, 1996]. AgentSpeak(L) can be viewed as a simplified, textual language of PRS. The proposal of AgentSpeak(L) was motivated by the question of how to relate implemented BDI systems to the BDI logics. Rao observes that the complexity of the code of implemented BDI systems has meant that the implemented systems have lacked a strong theoretical underpinning, i.e., the relation between these systems and BDI logics was not clear. While the issue of the relation between implemented BDI systems and BDI logics was not resolved in [Rao, 1996] and is still an open issue (see Section 5.2.1), the proposal of AgentSpeak(L) was followed by an increasing amount of research into agent programming languages.

In particular, we mention the proposals for the languages 3APL [Hindriks et al., 1999b, Hindriks, 2001] and GOAL [Hindriks et al., 2001], on which most of the work in this thesis builds. Like AgentSpeak(L), 3APL and GOAL are inspired by BDI theory. Other examples of such languages and frameworks are JACK [Winikoff, 2005] and Jadex [Braubach et al., 2005, Pokahr et al., 2005b]. JACK is an extension of Java [Gosling et al., 2000] with agent-oriented constructs, and Jadex is a BDI reasoning engine implemented in Java that provides an execution environment and an API. Another language that has several agent-oriented extensions, is MetateM [Fisher, 2006]. MetateM is a programming language based on direct execution of temporal logic statements. An extension inspired by BDI was proposed in [Fisher, 1997].

In the early days of agent programming research, the term agent programming was synonymous with agent programming inspired by BDI theory. However, as the agent systems field grew, many other approaches to agents and agent programming emerged. For example, there is a large body of work taking an algorithmic - rather than a programming - approach towards multi-agent issues such as teamwork or negotiation. Also, there are agent programming frameworks that are not based on BDI theory, such as the IMPACT framework [Dix and Zhang, 2005]. IMPACT builds on logic programming, and focuses on the development of a multi-agent platform that is able to deal with heterogenous and distributed data, and that can be realized on top of arbitrary legacy code.

In order to distinguish agent programming frameworks that are based on BDI from those that are not, many use the term BDI agent programming. However,

the precise relation between such frameworks and BDI theory is generally not clear. Also, notions that are used in such frameworks are usually not precisely the notions of beliefs, desires and intentions. That is, related notions such as plans, goals, events, and capabilities are frequently used instead of, or in addition to the BDI notions. The notions in this wide spectrum have also been referred to as *cognitive* notions. Therefore, we prefer to use the term *cognitive agent programming* for approaches such as ours, that are inspired by BDI theory but also use related cognitive notions.

In particular, the cognitive notions that we generally use are the notions of *beliefs*, *goals*, and *plans*. The idea is that an agent should try to reach its goals by executing appropriate plans, given its beliefs. Beliefs thus form the informational component of the agent, goals form the motivational component, and plans form the procedural component.

Most cognitive agent programming languages have at least an informational and a procedural component. The term "beliefs" is used for the informational component in most of these languages. As for the procedural component, several languages also use the term "intentions" instead of or in addition to "plans". Regarding the motivational component, the term "goals" tends to appear more often in cognitive agent programming frameworks than the term "desires". We refer to Section 5.1.2 for a short discussion on the difference between goals and desires.

The aspects of cognitive agent programming on which we focus in this thesis, are the representation of goals (Part I), and a particular aspect of the execution of plans, i.e., plan revision (Part II). In Part III, we address some software engineering aspects. In Chapter 2, we present a simple cognitive agent programming language that incorporates beliefs, goals, plans, and a mechanism for plan revision. At the end of that chapter, we present a more detailed overview of this thesis.

1.2 Formal Semantics

Cognitive agent programming is one of the pillars on which this thesis rests. Formal semantics of programming languages is the other.

Research on formal semantics[3] of programming languages is concerned with the rigorous mathematical study of the meaning of programming languages and models of computation. When defining a formal semantics for a programming language, the constructs of the language are related to objects in some domain of interpretation. That is, formal semantics is a way to give a precise specification of what it means to execute a program.

Formal semantics can be contrasted with the informal semantic descriptions as can be found in reference manuals and language standards. Such informal

[3]In other chapters, we generally omit the adjective "formal".

descriptions are less precise by nature, and are based primarily on implementation techniques and intuitions. The use of formal semantics as a technique for describing the meaning of programming languages has several advantages over an informal approach.

An important advantage of using formal semantics as a tool for designing programming languages is that *semantic issues or problems* are more likely to be *identified*. In the foreword of [Tennent, 1991, page xvii], Reynolds puts it as follows: "The truth of the matter is that putting languages together is a very tricky business. When one attempts to combine language concepts, unexpected and counterintuitive interactions arise. At this point, even the most experienced designer's intuition must be buttressed by a rigorous definition of what the language means."[4] In this thesis, we will see several cases in which the fact that formal semantics forces one to be precise, brings to the surface issues that could otherwise have been overlooked easily.

Another benefit of using formal semantics is that it provides a basis for *comparing* different languages. Without a precise description of the meaning of a language, it is difficult to make any claims on the exact relation between different languages. With the emergence of more and more cognitive agent programming languages, this becomes even more important if one wants to understand whether seemingly different constructs really *are* different. Such understanding can facilitate convergence towards one or a small number of cognitive agent programming languages.

Further, formal semantics is essential if one wants to do - especially deductive - *verification*. That is, if one wants to prove that a proof system for a programming language is sound and complete, one will have to define the semantics of this programming language (see also [de Bakker, 1980]). Otherwise, it becomes problematic to define when a specification is true for a program, which is necessary for proving soundness and completeness. In this thesis, we will be concerned with verification mainly in Chapter 7.

In the context of cognitive agent programming, formal semantics is a necessary prerequisite for establishing a formal relation between cognitive agent programming languages and BDI logics. As noted in Section 1.1, it is precisely this which has motivated Rao to propose the programming language AgentSpeak(L) as an alternative to the BDI architecture PRS.

Finally, we mention the following as a reason for the use of formal semantics. Generally, formal semantics is regarded as an area of theoretical computer science, and is consequently viewed as a field that does not have an immediate connection with the everyday programming practice. In particular, in the field of cognitive agent programming, approaches such as ours are generally regarded as theoretical, whereas approaches that, e.g., build on Java and do not have formal semantics, are regarded as practical.

[4]Note that "putting together" means "constructing" or "creating", rather than "combining".

While this may be true to a certain extent, we like to argue that the uptake of cognitive agent programming languages in mainstream software engineering can be facilitated by using formal semantics in the design of these languages. The reason for this is that the use of formal semantics generally helps in getting a better understanding of the language. A better understanding helps to identify the essence of a language, which can contribute to the design of a simpler language that captures these essential aspects.[5] We believe that clarity and simplicity are features that greatly increase the chances of a language being used in practice. As formal semantics can be a tool for achieving clarity and simplicity, formal semantics can be viewed as a tool for advancing the use of cognitive agent programming languages in practice.

In summary, formal semantics provides a solid foundation for the development of programming languages, and we hope that this thesis provides such a foundation for the further development of *cognitive agent* programming languages.

[5]Note that we do not argue that simpler is always better. Rather, we argue that a language should not be *unnecessarily* complicated. However, if more complicated features achieve a certain desired level of expressiveness, the language designer should not refrain from adding these features.

Chapter 2

Setting the Stage

The aim of this chapter is to set the stage for the rest of this thesis. We introduce a simple cognitive agent programming language and show how we can concretize various mental attitudes in a programming language. This language combines elements of simplified versions of the cognitive agent programming languages 3APL [Hindriks et al., 1999b] and GOAL [Hindriks et al., 2001], and resembles the language Dribble [van Riemsdijk et al., 2003b]. In the rest of this thesis, we will be concerned with a further investigation of certain aspects of this language.

2.1 Syntax

2.1.1 Beliefs, Goals, and Plans

The basic components of our language are *beliefs*, *goals*, and *plans*. Beliefs are represented using a so-called *belief base*, describing the information an agent has about the world and internal information. In this thesis, a belief base will consist of a set of propositional formulas. In principle, one could use other knowledge representation languages, such as first order languages (see, e.g., [Hindriks et al., 1999b, Dastani et al., 2004] and [Hindriks, 2001, Chapter 2]). We choose to use a propositional language, because its relative simplicity allows us to focus on the essence of the aspects of cognitive agent programming languages that we are concerned with in this thesis. Goals are represented using a goal base, which also consists of a set of propositional formulas. These formulas represent the situations the agent wants to achieve.

Definition 2.1 *(belief base and goal base)* Assume a propositional language $\mathcal{L}$ with typical formula ϕ and the connectives $\wedge$ and $\neg$ with the usual meaning. An agent's belief base is typically denoted by σ, and is a subset of $\mathcal{L}$, i.e., $\sigma \subseteq \mathcal{L}$. Similarly, an agent's goal base is typically denoted by γ, where $\gamma \subseteq \mathcal{L}$.

In this thesis, we will often use Σ to denote the set of belief bases, where $\Sigma = \wp(\mathcal{L})$. In order to refer to the beliefs and goals of an agent, we generally use so-called belief and goal formulas as defined below. These formulas can be used to express that the agent has or does not have certain beliefs or goals.

Definition 2.2 *(belief and goal formulas)* The belief formulas $\mathcal{L}_{\mathsf{B}}$ with typical element β and the goal formulas $\mathcal{L}_{\mathsf{G}}$ with typical element κ are defined as follows, where $\phi \in \mathcal{L}$.

$$\beta ::= \top \mid \mathbf{B}\phi \mid \neg\beta \mid \beta_1 \wedge \beta_2$$
$$\kappa ::= \top \mid \mathbf{G}\phi \mid \neg\kappa \mid \kappa_1 \wedge \kappa_2$$

Note that the **B** and **G** operators cannot be nested, i.e., formulas of the form $\mathbf{BG}\phi$ or $\mathbf{BB}\phi$ are not part of the languages.

Plans are the agent's means for achieving its goals. A plan is a sequence of *basic actions* and so-called *abstract plans*. As in 3APL and GOAL, basic actions can be executed, resulting in a change of the agent's belief base. The execution of actions may also cause changes in the agent's environment through some interface (see also [Hindriks, 2001, Chapter 2]). The interaction of an agent with its environment is however not considered in this thesis where formal semantics is concerned (see Sections 2.2 and 2.5). An abstract plan cannot be executed directly in the sense that it updates the belief base of an agent. Abstract plans, on the other hand, serve as an abstraction mechanism like procedures in imperative programming. If a plan consists of an abstract plan, this abstract plan could be transformed into basic actions through reasoning rules, which will be introduced below.

Definition 2.3 *(plan)* Assume that a set $\mathsf{BasicAction}$ with typical element a is given, together with a set $\mathsf{AbstractPlan}$ with typical element p. Let $c \in (\mathsf{BasicAction} \cup \mathsf{AbstractPlan})$. Then the set of plans Plan with typical element π is defined as follows.

$$\pi ::= a \mid p \mid c; \pi$$

The language of plans can be extended in a straightforward way to incorporate, e.g., constructs for programming conditional choice (if-then-else) or loops (while). These extensions are however not needed for the investigations carried out in this thesis. The reason for defining sequentially composed plans as $c;\pi$, rather than as $\pi_1;\pi_2$ as was, e.g., done in [Hindriks et al., 1999b], will be explained in Section 2.2 (Remark 2.2).

2.1.2 Reasoning Rules

The idea is that agents in our language try to achieve their goals by means of executing plans. In order to allow the programmer to specify which plan an agent should execute for which goals, we introduce a language construct called *plan selection rule*. Plan selection rules are of the form $\kappa \mid \beta \Rightarrow \pi$. This rule represents

that the agent may select plan π if the belief condition β and the goal condition κ hold. These rules were first introduced in [van Riemsdijk et al., 2003b], and are derivative from rules introduced in [Hindriks et al., 2001].

Besides plan selection rules, agents in our language also have so-called *plan revision rules*. Plan revision rules are of the form $\pi_h \mid \beta \rightsquigarrow \pi_b$. Informally, this rule expresses that if the agent has a plan π_h, and believes β to be the case, it may replace the plan π_h by the plan π_b. These rules can thus be used by the agent to revise its plan at run-time. Plan revision rules were first introduced in [Hindriks et al., 1999b], and were termed "practical reasoning rules" in that paper.

Definition 2.4 *(reasoning rules)* The sets of plan selection rules $\mathcal{R}_{\mathsf{PS}}$ and plan revision rules $\mathcal{R}_{\mathsf{PR}}$ are defined as follows.[1]

$$\begin{array}{lcll} \mathcal{R}_{\mathsf{PS}} & = & \{\kappa \mid \beta \Rightarrow \pi & : \ \kappa \in \mathcal{L}_{\mathsf{G}},\ \beta \in \mathcal{L}_{\mathsf{B}},\ \pi \in \mathsf{Plan}\} \\ \mathcal{R}_{\mathsf{PR}} & = & \{\pi_h \mid \beta \rightsquigarrow \pi_b & : \ \beta \in \mathcal{L}_{\mathsf{B}},\ \pi_h, \pi_b \in \mathsf{Plan}\} \end{array}$$

2.1.3 An Agent

To program an agent in our language means to specify its initial beliefs and goals, and to write sets of plan selection rules and plan revision rules. Also, an agent contains a specification of how the belief base of the agent is updated if basic actions are executed. For reasons of technical convenience and as done in other papers on similar languages (see, e.g., [Hindriks et al., 1999b]), we assume for this a function $\mathcal{T}$ that takes a belief base and a basic action, and yields a new belief base resulting from the execution of the action.

This function could be defined using advanced techniques from the field of belief revision, or, if the belief base has a simple structure, simpler methods could be used. Investigating belief revision techniques is however not the topic of this thesis. In Chapter 10, we do present a simple method for specifying belief update through action execution, which is also the one used in implementations of 3APL.

Definition 2.5 *(agent)* An agent is a tuple $\langle \sigma_0, \gamma_0, \mathsf{PS}, \mathsf{PR}, \mathcal{T} \rangle$, where $\sigma_0 \subseteq \mathcal{L}$ is the initial belief base, $\gamma_0 \subseteq \mathcal{L}$ is the initial goal base, $\mathsf{PS} \subseteq \mathcal{R}_{\mathsf{PS}}$ and $\mathsf{PR} \subseteq \mathcal{R}_{\mathsf{PR}}$ are the plan selection and plan revision rules, respectively, and $\mathcal{T} : (\mathsf{BasicAction} \times \Sigma) \rightarrow \Sigma$ is a partial function where $\Sigma = \wp(\mathcal{L})$ is the set of belief bases, expressing how belief bases are updated through basic action execution.

Note that the agent is not endowed with an initial plan. The idea is that an agent should use its plan selection rules to adopt plans to reach its goals. Further, we use the term agent here to refer to the tuple containing the initial

[1] We use the notation $\{\ldots : \ldots\}$ instead of $\{\ldots \mid \ldots\}$ to define sets, to prevent confusing usage of the symbol $\mid$ in this definition.

belief base and goal base, and reasoning rules and belief update function. It might be considered more appropriate to use the term "agent program" for this tuple, and to use the term "agent" for the corresponding computational entity when it is executing. However, in this thesis we will loosely use the term agent in both cases.

2.2 Semantics

In the previous section, we have defined the components of an agent. In this section, we specify what it means to *execute* such an agent, i.e., we specify its semantics. In order to do this, we introduce the notion of a *configuration*. Configurations are used to represent the (components of the) agent at each moment during execution. The reasoning rules and the belief update function do not change during computation. We will thus generally not include these in configurations, for reasons of presentation. Rather, configurations consist of a belief base, a goal base, and a plan. Configurations in the context of cognitive agent programming languages are sometimes also called *mental states* (see Chapter 6 and [Hindriks et al., 1999b, van Riemsdijk et al., 2003b]).

Below, we formally define configurations. The third component of a configuration represents the plan of the agent. This may either be a plan consisting of a sequence of basic actions and abstract plans as specified in Definition 2.3, or it may be an empty plan, which is denoted by ϵ. The empty plan is introduced for technical convenience, which will be further explained below (Remark 2.1).

Definition 2.6 *(configuration)* A configuration is a tuple $\langle \sigma, \gamma, \pi \rangle$ where $\sigma \subseteq \mathcal{L}$, $\gamma \subseteq \mathcal{L}$, and $\pi \in (\mathsf{Plan} \cup \{\epsilon\})$. If $\langle \sigma_0, \gamma_0, \mathsf{PS}, \mathsf{PR}, \mathcal{T} \rangle$ is an agent, then $\langle \sigma_0, \gamma_0, \epsilon \rangle$ is the initial configuration of the agent.

Before defining what it means to execute an agent, we need to specify the semantics of belief and goal formulas, since the application of reasoning rules depends on this. Belief and goal formulas are evaluated in a configuration. The semantics specifies that $\mathbf{B}\phi$ is true in a configuration if ϕ follows from the belief base of the agent. A formula $\mathbf{G}\phi$ is true if ϕ follows from the goal base, and ϕ does not follow from the belief base. The idea is, that an agent should not have something as a goal which it already believes to be achieved. These kinds of goals are generally called *achievement goals*, and these are the goals that we mostly consider in this thesis. We refer to Chapter 5 for a further discussion of various kinds of goals.

Definition 2.7 *(semantics of belief and goal formulas)* Let $\phi \in \mathcal{L}$ and let $\models$ be the standard entailment relation for $\mathcal{L}$. Further, let $\langle \sigma, \gamma, \pi \rangle$ be an agent configuration, let $\beta, \beta_1, \beta_2 \in \mathcal{L}_{\mathsf{B}}$ and let $\kappa, \kappa_1, \kappa_2 \in \mathcal{L}_{\mathsf{G}}$. The semantics $\models_{\mathcal{L}_{\mathsf{B}}}$ and

$\models_{\mathcal{L}_{\mathsf{G}}}$ of belief and goal formulas, respectively, are then as defined below.

$$
\begin{array}{lcl}
\langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{B}}} \top & & \\
\langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{B}}} \mathbf{B}\phi & \Leftrightarrow & \sigma \models \phi \\
\langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{B}}} \neg\beta & \Leftrightarrow & \langle \sigma, \gamma, \pi \rangle \not\models_{\mathcal{L}_{\mathsf{B}}} \beta \\
\langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{B}}} \beta_1 \wedge \beta_2 & \Leftrightarrow & \langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{B}}} \beta_1 \text{ and } \langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{B}}} \beta_2
\end{array}
$$

$$
\begin{array}{lcl}
\langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{G}}} \top & & \\
\langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{G}}} \mathbf{G}\phi & \Leftrightarrow & \gamma \models \phi \text{ and } \sigma \not\models \phi \\
\langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{G}}} \neg\kappa & \Leftrightarrow & \langle \sigma, \gamma, \pi \rangle \not\models_{\mathcal{L}_{\mathsf{G}}} \kappa \\
\langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{G}}} \kappa_1 \wedge \kappa_2 & \Leftrightarrow & \langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{G}}} \kappa_1 \text{ and } \langle \sigma, \gamma, \pi \rangle \models_{\mathcal{L}_{\mathsf{G}}} \kappa_2
\end{array}
$$

It is important to note that these semantics have the property that the formulas $\mathbf{B}\neg\phi$ and $\neg\mathbf{B}\phi$ (and similarly for $\mathbf{G}\neg\phi$ and $\neg\mathbf{G}\phi$) are not equivalent. That is, $\mathbf{B}\neg\phi$ holds in a configuration c with belief base σ iff $\sigma \models \neg\phi$, and $\neg\mathbf{B}\phi$ holds in c iff $c \not\models_{\mathcal{L}_{\mathsf{B}}} \mathbf{B}\phi$, which holds iff $\sigma \not\models \phi$. The formula $\mathbf{B}\neg\phi$ thus expresses that $\neg\phi$ should follow from the belief base (in the standard propositional logic sense), while $\neg\mathbf{B}\phi$ expresses that ϕ should *not* follow. In general, this is not equivalent. Consider, e.g., a configuration c with belief base $\{q\}$. In this case, $\mathbf{B}(\neg p)$ does not hold in c, while $\neg\mathbf{B}p$ *does* hold. If the belief base is consistent, we do have that $\neg\mathbf{B}\phi$ holds if $\mathbf{B}\neg\phi$ holds.

Further, we make a remark with respect to the semantics of goal formulas. From the perspective of the semantics of goal formulas, the goal bases $\gamma_1 = \{p, q\}$ and $\gamma_2 = \{p \wedge q\}$, for example, are equivalent. Consider, e.g., the formula $\mathbf{G}(p \wedge q)$. This formula is true in configurations with goal base γ_1 (assuming that $p \wedge q$ is not believed by the agent), and in configurations with goal base γ_2. This is because anything that follows from γ_1 (in the standard propositional logic sense) also follows from γ_2, and vice versa. However, these two goal bases are not equivalent when considering the semantics of the execution of an agent, which will be introduced below. The idea is that γ_2 represents that the agent wants to achieve a situation in which p and q hold *at the same time*, while in the case of γ_1, the agent does not necessarily has to strive for a situation in which both p and q hold. It may be the case that it first achieves p, and then q. These ideas are reflected in the specification of when goals should be removed from the goal base (see Definition 2.9 below).

We now move on to defining the semantics of executing an agent in our language. This semantics is defined using a transition system [Plotkin, 1981]. A transition system for a programming language consists of a set of axioms and transition rules for deriving transitions for this language. A transition is a transformation of one configuration into another and it corresponds to a single computation step. In the transition rules below, we assume an agent with a set of plan selection rules PS, a set of plan revision rules PR, and a belief update function $\mathcal{T}$.

An agent's configuration may change through the execution of an action from the agent's plan, or through the application of a reasoning rule. Corre-

spondingly, the transition system consists of three transition rules, i.e., one for executing an action, one for applying a plan selection rule, and one for applying a plan revision rule.

Below, we present the transition rule for basic action execution. In that transition rule, we use a function $\bullet : \mathsf{Plan}' \rightarrow (\mathsf{Plan}' \rightarrow \mathsf{Plan}')$, where Plan' is defined as $\mathsf{Plan} \cup \{\epsilon\}$. This function takes two plans, that may also be empty plans, and concatenates these.

Definition 2.8 *(plan concatenation)* Let $\pi' \in \mathsf{Plan}'$, $\pi \in \mathsf{Plan}$, and let $c \in (\mathsf{BasicAction} \cup \mathsf{AbstractPlan})$. The function $\bullet : \mathsf{Plan}' \rightarrow (\mathsf{Plan}' \rightarrow \mathsf{Plan}')$ which concatenates two plans, is then defined as follows, using infix notation.

$$\begin{array}{lcl} \epsilon \bullet \pi' & = & \pi' \\ \pi' \bullet \epsilon & = & \pi' \\ c \bullet \pi & = & c; \pi \\ c; \pi \bullet \pi' & = & c; (\pi \bullet \pi') \end{array}$$

We will explain why we need this function (Remark 2.1), after presenting the definition of the transition rule for basic action execution. Basic actions update the belief base of an agent if they are executed, as specified through the partial function $\mathcal{T}$, where $\mathcal{T}(a, \sigma)$ returns the result of updating belief base σ by performing action a. The fact that $\mathcal{T}$ is a partial function represents that an action may not be executable in some belief states. Goals that are reached through execution of an action are removed from the goal base. After execution of an action, the action is removed from the plan.

Definition 2.9 *(action execution)* Let $a \in \mathsf{BasicAction}$ and let $\gamma' = \gamma \setminus \{\phi \in \gamma \mid \sigma' \models \phi\}$.

$$\frac{\mathcal{T}(a, \sigma) = \sigma'}{\langle \sigma, \gamma, a \bullet \pi \rangle \rightarrow \langle \sigma', \gamma', \pi \rangle}$$

Remark 2.1 *(empty plan and plan concatenation)* The reason that we have defined configurations such that the plan component may be either a plan from the set Plan *or* an empty plan, is as follows. Because π in the transition rule above may also be the empty plan ϵ, the transition rule specifies two variants of executing an action: the plan $a \bullet \pi$ may consist of an action a followed by a non-empty sequence of actions and abstract plans (in which case $\pi \in \mathsf{Plan}$), or the action a may be the only action of the plan (in which case $\pi = \epsilon$). If we would not have allowed π to be the empty plan, we would have had to introduce another transition rule, that specifies how to execute an action if the action is the only action of the plan, i.e., if the plan component of the configuration is of the form a.

A consequence of this is that we can, strictly speaking, not use the sequential composition operator for specifying the configuration on the left-hand side of

the transition, i.e., by specifying the configuration as $\langle \sigma, \gamma, a; \pi \rangle$. Following Definition 2.3, the sequential composition operator is defined on basic actions and non-empty plans from the set Plan. The concatenation operator solves our problem, because this operator *is* defined also on empty plans.

Alternatively, we could have introduced empty plans in the language of plans of Definition 2.3. This would however have allowed us to define, e.g., a plan of the form $\epsilon; \epsilon; \epsilon; a$, from which our language would hardly benefit. Also, this would not have provided the advantage of having to specify only one transition for the execution of basic actions, since a plan $a; \epsilon$ would then consist of two elements, rather than only of a, as in the case of $a \bullet \epsilon$. On the contrary, we would need to specify another transition for the execution of empty plans, in order to get rid of the empty plans in, e.g., the plan $\epsilon; \epsilon; \epsilon; a$.

Having explained all this, we will not be so strict in the rest of this thesis.

A plan selection rule is applicable in a configuration if the goal condition and belief condition of the rule are true in that configuration. Further, it can only be applied if the plan of the agent is empty. The idea is, that an agent can only select a new plan if it has finished executing its old plan. In the resulting configuration, the plan becomes equal to the consequent of the plan selection rule.

Definition 2.10 *(plan selection)*

$$\frac{\kappa \mid \beta \Rightarrow \pi \in \mathsf{PS} \qquad \langle \sigma, \gamma, \epsilon \rangle \models_{\mathcal{L}_\mathsf{G}} \kappa \qquad \langle \sigma, \gamma, \epsilon \rangle \models_{\mathcal{L}_\mathsf{B}} \beta}{\langle \sigma, \gamma, \epsilon \rangle \rightarrow \langle \sigma, \gamma, \pi \rangle}$$

We could have defined configurations as containing a plan base consisting of a *set* of plans, rather than of a single plan, as done in, e.g., [Hindriks et al., 1999b, Dastani et al., 2004]. In that case, plan selection rules could be used to *add* plans to the plan base. Such a choice would however introduce added complexity to the semantics of plan execution, i.e., one would have to define how plans are executed concurrently, and it would call for a discussion on possible interferences between plans that are executed concurrently. Except in Chapter 4, we will avoid these additional complications throughout this thesis by having the plan base consist of only one plan.

A plan revision rule can be applied in a configuration if the head of the rule is equal to a prefix of the plan in the configuration. The application of the rule results in the revision of the plan, such that the prefix equal to the head of the rule is replaced by the plan in the body of the rule. A rule $a; b \mid \top \rightsquigarrow c$ can for example be applied to the plan $a; b; c$, yielding the plan $c; c$. The belief base is not changed through plan revision. As in the transition rule for action execution (Definition 2.9), we use the concatenation operator.

Definition 2.11 *(plan revision)*

$$\frac{\pi_h \mid \beta \rightsquigarrow \pi_b \in \mathsf{PR} \qquad \langle \sigma, \gamma, \pi_h \bullet \pi \rangle \models_{\mathcal{L}_\mathsf{B}} \beta}{\langle \sigma, \gamma, \pi_h \bullet \pi \rangle \rightarrow \langle \sigma, \gamma, \pi_b \bullet \pi \rangle}$$

Remark 2.2 *(structure of plans)* We are now in a position to explain why we have chosen to define sequentially composed plans as $c;\pi$ (where c is an atomic plan element, i.e., a basic action or an abstract plan), rather than as $\pi_1;\pi_2$. If we would have chosen the latter option, the configuration on the left-hand side of the transition for plan revision could have been defined as $\langle \sigma, \gamma, \pi_h;\pi\rangle$ (ignoring for the moment that we might want to allow π to be the empty plan).

Given, e.g., a plan revision rule $a;b \mid \top \rightsquigarrow c$, one could then argue that it is not clear whether this rule is applicable to, e.g., the plan $a;b;c$ (where a, b, and c are basic actions). The reason is that this plan could be viewed as the sequential composition of a and $b;c$, or as the sequential composition of $a;b$ and c. If the plan is viewed as $(a;b);c$ (where we introduce brackets to indicate how the plan is composed), the rule can be applied. If, however, the plan is viewed as $a;(b;c)$, the rule cannot be applied, since this plan cannot be decomposed into the plan $a;b$ and the plan c. The idea is, however, that the rule should be applicable to both plans, regardless of how they are composed.

Our solution to this problem is to define an unambiguous grammar of plans by defining sequentially composed plans as $c;\pi$. The plan $a;b;c$ can then only be viewed as the plan $a;(b;c)$. Now, we have to make sure that we define the semantics such, that the rule $a;b \mid \top \rightsquigarrow c$ is applicable to this plan. Given this definition of plans, the desired semantics can be achieved easily by using the concatenation operator, as done in Definition 2.11. The transition rule for plan revision specifies that the rule $a;b \mid \top \rightsquigarrow c$ can be applied to any plan resulting from concatenating $a;b$ and some plan π. In particular, the rule can be applied to the plan resulting from the concatenation of $a;b$ and c, i.e., to the plan $a;(b;c)$.

Having explained this, we will not always be so strict in this thesis.

The semantics of an agent in our language is derived directly from the transition relation $\rightarrow$. It can be defined in various ways. One way of defining the semantics, is to define it as a set of so called computation runs.

Definition 2.12 *(semantics)* A computation run for an agent $\langle \sigma_0, \gamma_0, \mathsf{PS}, \mathsf{PR}, \mathcal{T}\rangle$ is a finite or infinite sequence $s_0, \ldots, s_n$ or $s_0, \ldots$ such that s_i for $i \geq 0$ are configurations, s_0 is the initial configuration of the agent, i.e., $s_0 = \langle \sigma_0, \gamma_0, \epsilon\rangle$ (see Definition 2.6), and $\forall_{i>0}: \; s_{i-1} \rightarrow s_i$ is a transition in the transition system for the agent. For finite computation runs $s_0, \ldots, s_n$ it should be the case that there is no transition $s_n \rightarrow s_{n+1}$ for some s_{n+1}. The semantics of the agent is then defined as the set of all computation runs for this agent.

2.3 Example

We present a simple example in order to illustrate how the components of the language may be used, and to further clarify the semantic definitions (Section

2.3.1). In Section 2.3.2, we use the example to argue that goals and plan revision can be used for programming flexibly behaving agents.

2.3.1 Building a Tower

The example is adapted from [van Riemsdijk, 2002], and was also used in [van Riemsdijk et al., 2003b, Dastani et al., 2004]. Our example agent has to solve the problem of building a tower of blocks. The blocks have to be stacked in a certain order: block C has to be on the floor (Fl), B should be on C, and block A should be on B. Initially, the blocks A and B are on the floor, while C is on A.

The only action an agent can take, is to move a block x from some block y to another block z ($move(x, y, z)$). Variables are used to ease the specification of the example agent, and should be instantiated with the relevant arguments. That is, the action $move(x, y, z)$ represents the set of actions

$$\{move(x, y, z) \ : \ x \in \{A, B, C\} \text{ and } y, z \in \{A, B, C, Fl\} \text{ and } x \neq y \neq z\}.$$

The $move(x, y, z)$ action is enabled only if the block to be moved (x) and the block to which x is moved (z) are clear. The result of the action is, that x is on z and not on y, block y becomes clear and block z is not clear anymore (assuming that z is not the floor, because the floor is always clear). Let the agent's initial belief base σ_0, goal base γ_0, plan selection rules PS, and plan revision rules PR, be as defined below. The function $\mathcal{T}$ will be specified in the sequel.

$$\begin{array}{lcl}
\sigma_0 & = & \{on(C, A),\ on(A, Fl),\ on(B, Fl),\ clear(B),\ clear(C),\ clear(Fl)\} \\
\gamma_0 & = & \{on(A, B) \wedge on(B, C) \wedge on(C, Fl)\} \\
\mathsf{PS} & = & \{\mathbf{G}(on(x, z)) \mid \mathbf{B}(on(x, y)) \Rightarrow move(x, y, z) \ : \\
 & & \qquad x \in \{A, B, C\} \text{ and } y, z \in \{A, B, C, Fl\} \text{ and } x \neq y \neq z\} \\
\mathsf{PR} & = & \{move(x, y, z) \mid \mathbf{B}(on(u, x)) \rightsquigarrow move(u, x, Fl); move(x, y, z) \ : \\
 & & \qquad x, u \in \{A, B, C\} \text{ and } y, z \in \{A, B, C, Fl\} \text{ and } x \neq y \neq z\} \cup \\
 & & \{move(x, y, z) \mid \mathbf{B}(on(u, z)) \rightsquigarrow move(u, z, Fl); move(x, y, z) \ : \\
 & & \qquad x, u \in \{A, B, C\} \text{ and } y, z \in \{A, B, C, Fl\} \text{ and } x \neq y \neq z\}
\end{array}$$

The goal rules are used to derive the $move(x, y, z)$ action that should be executed to fulfil a goal $on(x, z)$. The preconditions of the move action are not checked in these rules, so it is possible that the derived action cannot be executed in a particular configuration. The plan revision rules can then be used to create a configuration in which this action *can* be executed. Note that the goal rules are used to select an action to fulfil a single proposition of the form $on(x, z)$. The initial goal base however contains a conjunction of $on(x, z)$ propositions. The goal rule is applicable to this conjunction, because a formula $\mathbf{G}\phi$ is true if ϕ is a logical consequence of the goal base (see Definition 2.7).

We assume that the function $\mathcal{T}$, when applied to basic action $move(x, y, z)$ and a belief base σ has the following properties. The function is defined iff

σ entails $clear(x) \wedge clear(z) \wedge on(x, y)$. The result is a belief base that entails $on(x, z) \wedge clear(y)$ and that does not entail $on(x, y)$ and $clear(z)$ if $z \neq Fl$. If $z = Fl$, the resulting belief base entails $on(x, z) \wedge clear(y) \wedge clear(z)$ and does not entail $on(x, y)$. This describes the changes to the propositions $on(x, y), on(x, z), clear(y)$ and $clear(z)$ which are caused by the action $move(x, y, z)$. Below, we specify which propositions are not changed by the action $move(x, y, z)$, assuming that the function $\mathcal{T}$ is defined on action $move(x, y, z)$ and belief base σ, i.e., let $\mathcal{T}(move(x, y, z), \sigma) = \sigma'$.

$$\begin{array}{lcll} \sigma \models on(u, v) & \wedge & u \neq x & \Rightarrow \quad \sigma' \models on(u, v) \\ \sigma \models clear(w) & \wedge & w \neq z & \Rightarrow \quad \sigma' \models clear(w) \end{array}$$

In the initial configuration of the agent $\langle \sigma_0, \gamma_0, \epsilon \rangle$, three possible plan selection rules could be applied: $x = A$, $y = Fl$, $z = B$ or $x = B$, $y = Fl$, $z = C$ or $x = C$, $y = A$, $z = Fl$ (yielding $move(A, Fl, B), move(B, Fl, C)$ or $move(C, A, Fl)$). Suppose the first is chosen. After application of this plan selection rule, the plan of the agent becomes equal to the plan in the consequent of the plan selection rule, resulting in the following configuration.

$$\begin{array}{lcl} \sigma_1 & = & \{on(A, Fl), on(B, Fl), on(C, A), clear(B), clear(C), clear(Fl)\} \\ \gamma_1 & = & \{on(A, B) \wedge on(B, C) \wedge on(C, Fl)\} \\ \pi_1 & = & move(A, Fl, B) \end{array}$$

The plan cannot be executed because the preconditions of the action are not satisfied in this configuration (there is a block on A). The plan selection rule cannot be applied because the plan of the agent is not empty. The only applicable rule is the plan revision rule $move(A, Fl, B) \mid \mathbf{B}(on(C, A)) \rightsquigarrow move(C, A, Fl); move(A, Fl, B)$.

$$\begin{array}{lcl} \sigma_2 & = & \{on(A, Fl), on(B, Fl), on(C, A), clear(B), clear(C), clear(Fl)\} \\ \gamma_2 & = & \{on(A, B) \wedge on(B, C) \wedge on(C, Fl)\} \\ \pi_2 & = & move(C, A, Fl); move(A, Fl, B) \end{array}$$

The only option is to execute the first action of the plan, resulting in a changed belief base that has the property as specified below.

$$\begin{array}{lcl} \sigma_3 & \models & on(A, Fl) \wedge on(B, Fl) \wedge on(C, Fl) \wedge clear(A) \wedge clear(B) \wedge \\ & & clear(C) \wedge clear(Fl) \\ \gamma_3 & = & \{on(A, B) \wedge on(B, C) \wedge on(C, Fl)\} \\ \pi_3 & = & move(A, Fl, B) \end{array}$$

In this configuration, the action $move(A, Fl, B)$ is executed. In the resulting configuration, the only (logical consequence of the) goal which is not satisfied, is $on(B, C)$. A plan is constructed to move B onto C: first A is moved onto the floor, then B is moved onto C. The only goal which is not satisfied is $on(A, B)$. The action $move(A, Fl, B)$ is selected using a plan selection rule and

then executed. This results in the following final configuration in which the goal is reached and thus removed from the goal base.

$$\begin{array}{lcl} \sigma_F & \models & on(A,B) \wedge on(B,C) \wedge on(C,Fl) \wedge clear(A) \wedge clear(Fl) \\ \gamma_F & = & \emptyset \\ \pi_F & = & \epsilon \end{array}$$

This example execution shows that the agent can reach its initial goal. The number of actions that the agent needs to execute before it reaches its goal, is partly dependent on which choices are made if multiple plan selection rules are applicable. As was shown, three plan selection rules are for instance applicable in the initial configuration.

2.3.2 Goals and Plan Revision: Programming Flexible Agents

Goals and plan revision form two important aspects of the presented cognitive agent programming language. While these aspects can be studied relatively independently as we do in this thesis (see Section 2.4 for an overview), they can nevertheless be linked by viewing them as both facilitating the programming of *flexible* agents.

Agents need to have a certain level of flexibility, since they are often expected to operate in dynamic environments. The idea that goals can provide this added flexibility, can be explained using the tower building example (see Chapter 5 for a more elaborate discussion of advantages of goals).

From the transition rules of Section 2.2, we know that a goal is removed from the goal base only if the agent believes it has achieved the goal. Now consider the example tower building agent, which we will refer to as agent $\mathcal{A}$. Assume agent $\mathcal{A}$ is almost finished building its tower, i.e., block C is on the floor and B is on C, and A is on the floor. In order to finish the tower, it has to move block A onto block B. Now assume there is another agent in the environment, which at this point moves block B onto the floor, thereby destroying part of the tower. Also assume agent $\mathcal{A}$ observes this and updates its belief base accordingly.

Due to the fact that the goal $on(A,B) \wedge on(B,C) \wedge on(C,Fl)$ has not been removed from the goal base (because the tower has not been completed yet), the formula $\mathbf{G}(on(B,C))$ holds once more after the other agent has moved block B (even though $on(B,C)$ was previously believed, and therefore not a goal). Because of this, agent $\mathcal{A}$ can now apply the plan selection rule in order to move B onto C again. The fact that the agent has a goal base and that goals remain in the goal base until they are believed to be achieved, thus provides for flexibility in the sense that the agent will continue to try to achieve its goals, even if the environment changes in unhelpful ways, or if plans fail. While this kind of flexibility might also be achievable if programming in some general purpose language, the idea is that it can be beneficial that it is a built-in characteristic of the programming language.

This, of course, does not mean that using goals and plan selection rules will yield satisfactory results in every dynamic environment. For example, after the execution of a plan that has not achieved its desired objective (either because the programmer has not provided the right plan, or because there were changes in the environment that prevented the agent from reaching its goal), it may be the case that the agent finds itself in a situation in which there is no applicable plan selection rule. The agent may need to return to some initial situation, in order to be able to apply plan selection rules to try once more to achieve its goals. On the other hand, one could argue that the programmer should have made sure that the agent had the appropriate plan selection rules. That is, it is still the job of the programmer to make appropriate use of the components provided by the agent programming language, as is the case with any other programming language. Always providing the agent with the right plan selection rules can however be difficult if the environment gets "too unpredictable".

Another issue is that the agent may be left with part of a plan that is not executable anymore, because the environment has changed. In that case it needs to have a mechanism for getting rid of this old plan, and perhaps for changing the environment to allow it to start executing plans once again for achieving its goals. For this, plan revision rules could potentially be used.

Besides for cleaning up failed plans, plan revision rules can be used to provide for added flexibility in general. The idea is that plan revision rules can be used to revise an agent's plan, if the circumstances demand this. The belief condition can be used to specify these circumstances. In the tower building example for instance, plan revision rules are used to revise the agent's plan if a *move* action that is to be executed next, cannot be executed. The belief conditions of these rules specify the conditions under which a *move* action is not executable. In this example, the plan is revised in such a way that the desired *move* action can be executed eventually.

As in the case of goals and plan selection rules, it is the job of the programmer to specify appropriate plan revision rules. In order to be able to do this in an effective way, the environment cannot be too unpredictable. The programmer needs to be able to foresee up to a certain level where things might go wrong.

2.4 Overview of Thesis

In the previous sections, we have presented a cognitive agent programming language. So are we done now? Well... no. Several aspects of the presented language warrant a further investigation, parts of which will be carried out in the rest of this thesis. In particular, we will consider aspects related to *goals* (Part I), and aspects related to *plan revision* (Part II). Some *software engineering* aspects will be addressed in Part III.

2.4.1 Part I: Goals

In Part I, we investigate various ways of representing goals and of defining their semantics. In Chapter 3, we address, broadly speaking, the relation between the semantics of abstract plans and of goals. Abstract plans are sometimes viewed as the *subgoals* of an agent's plan, in the sense that an abstract plan p is sometimes interpreted as representing that the agent should achieve a state in which he believes p to be the case. In this light, we have found it interesting to study whether the properties of abstract plans correspond in some way to the properties of goals, and in particular the property that goals are not dropped until they are believed to be achieved (see Definition 2.7). Chapter 3 sheds some light on this issue, thereby clarifying whether aspects of the behavior of goals can be captured by a construct present in the plans of an agent.

In Chapter 4, we address issues regarding the semantics of goals, given that we want to allow the representation of conflicting goals. The semantics for goal formulas of Definition 2.7 is trivialized if the goal base of the agent is inconsistent. That is, if the goal base is inconsistent, the agent can derive $\mathbf{G}\bot$, and any other formula $\mathbf{G}\phi$. If one wants to allow the goal base to be inconsistent without trivializing the semantics of goal formulas if this is the case, this semantics will have to be adapted. In Chapter 4, we investigate such adaptations. Further, we propose an extension to the representation of goals, by proposing a construct for representing that goals may be conditional on beliefs or other goals. That is, by means of this construct one can represent that if the agent has a certain belief (or goal), it should also have a certain (other) goal. Given such a construct, we again investigate a semantics for goal formulas that allows the agent to have conflicting goals, without trivializing the semantics.

Concluding Part I, Chapter 5 contains a general discussion regarding literature addressing goals in the context of agent programming. We argue why we think goals are important in agent programming. Further, we identify important strands of research regarding the representation of goals in agent programming frameworks, thereby also showing how our work can be positioned with respect to other approaches.

2.4.2 Part II: Plan Revision

In Part II, we focus on plan revision. As our object of study, we take a language without goals (and therefore also without plan selection rules), and of course *with* plan revision rules. An important aspect of plan revision that calls for further investigation, and which is the aspect on which we focus in this thesis, is the semantics of the execution of plans in the presence of plan revision rules. As it turns out, this semantics is not compositional. This means, broadly speaking, that the semantics of a composed plan cannot be defined in terms of the semantics of the parts of which it is composed.

This is due to the presence of plan revision rules, and, in particular, to the

fact that the head of a plan revision rule may consist of a *composed* plan, rather than of an atomic plan. The issue of non-compositionality to which this gives rise, can be explained informally as follows.

Because of the presence of plan revision rules, the semantics of an atomic plan element cannot be considered in isolation. Put differently, the semantics of, e.g., a basic action a, depends on the actions that surround it. There may, e.g., be a plan revision rule with $a; b$ as the head, and one with $a; c$ as the head. Given a plan with a as the first action of the plan, i.e., a plan of the form $a; \pi$, the possible application of one of the plan revision rules depends on whether π is of the form $b; \pi'$ or of the form $c; \pi'$ (or of yet another form). When determining the semantics of the plan $a; \pi$, the action a can thus not be considered separately from π. If the semantics of plans would have been compositional, it would have been possible to consider a in isolation. The non-compositional nature of the semantics of plans gives rise to problems if one wants to prove properties of plan execution. If the semantics of plans is not compositional, one cannot prove properties of a composed plan by proving properties of the parts of which it is composed, which is what one would generally want to do.

This issue of the non-compositionality of the semantics of plans in the presence of plan revision rules is approached from three different perspectives in this thesis. In Chapter 6, we propose a meta-language on top of the agent programming language. This meta-language has constructs for specifying that an action should be executed, or that a plan revision rule should be applied. We show that it *is* possible to define a compositional semantics for this meta-language, and show that a particular meta-program is equivalent with the object-level semantics of the agent programming language. In Chapter 7, we take another approach and present a specialized dynamic logic for reasoning about plans in the presence of plan revision rules. This logic comes with a sound and complete axiomatization, and essentially circumvents the non-compositionality issue in a certain way. In Chapter 8, we approach the issue from again another angle, by restricting plan revision rules in such a way that the semantics of plans becomes compositional.

2.4.3 Part III: Software Engineering Aspects

Part III is different from the first two parts in that it does not address one particular aspect of the agent programming language presented in this chapter. The two chapters in this last part both address software engineering issues that concern the agent programming language as a whole.

Chapter 9 addresses modularization in cognitive agent programming languages. Modularization is widely recognized as a central issue in software engineering. A system which is composed of modules, i.e., relatively independent units of functionality, is called modular. When it comes to cognitive agent programming languages, one can think of various ways in which agent programs can be modularized. We discuss existing approaches to modularity in cognitive

agent programming, and propose a new kind of modularity, i.e., goal-oriented modularity. In goal-oriented modularity, the goals of an agent are taken as the basis for modularization. We present a formal semantics of goal-oriented modularity in the context of an agent programming language that is similar to the one presented in the present chapter.

In Chapter 10, we discuss a particular approach for prototyping an agent programming language similar to the one presented in this chapter. The approach we take is to implement the agent programming language in the Maude term rewriting language. Maude is based on the mathematical theory of rewriting logic. The language has been shown to be suitable both as a logical framework in which many other logics can be represented, and as a semantic framework, through which programming languages with semantics based on a transition system can be implemented in a rigorous way. We explore the usage of Maude in the context of agent programming languages, and argue that, since agent programming languages such as the one presented in this chapter have both a logical component (in the form of belief and goal formulas and their corresponding satisfaction relations) and a semantic component (in the form of the transition system), Maude is very well suited for prototyping such languages.

2.5 Important Issues We Do Not Address

A number of important issues related to cognitive agent programming languages are not addressed in detail in this thesis. In particular, even though we have argued that goals and plan revision can be linked by considering these as providing for added flexibility in the case of dynamic environments, we will in the rest of this thesis not consider aspects related to interaction with an environment. In particular, we will not address sensing, or issues related to the problem of making sure that the belief base accurately represents the environment. The main reason that we do not do this, is that it is not needed for the investigations into semantics of goals and plan revision that we carry out in this thesis. Incorporating the environment in the semantics of agent programming languages could be done by incorporating a model of the environment in the configurations. However, from a technical point of view this does not really differ from using a belief base as a model of the environment. Our theory would therefore hardly benefit from this, unless one would want to investigate issues related to sensing and updating of beliefs, which we do not.

Further, we do not aim to investigate advanced formalisms for representing an agent's beliefs. Throughout this thesis, the belief base will consist of a set of propositional formulas, and the semantics of belief formulas will be as in Definition 2.7.

Moreover, in this thesis we do not consider planning from first principles (see, e.g., [Fikes and Nilsson, 1971]), even though the vocabulary used in that field (beliefs, goals, plans) is similar to ours. The relation between planning from

first principles and plan revision (or agent programming in general) is discussed briefly in Section 7.1.

Furthermore, in this thesis we shall not dwell on the pragmatics of the discussed approach, but concentrate on theoretical issues.[2] In particular, we will not be concerned with agent-oriented design methodologies [Zambonelli et al., 2003, Bresciani et al., 2004].

Finally, we focus solely on aspects related to the programming of individual agents. Even though the slogan "There is no such thing as a single-agent system" is sometimes used, the origins of research into intelligent agents lie in research concerning single-agent issues (see [Wooldridge and Jennings, 1995] for an overview). In our view, research into single-agent aspects is an important part of agent systems research. In fact, we would like to propose the slogan "There is no multi-agent system without single agents".

[2] This sentence is adapted from a sentence in [de Bakker, 1980, page 127].

PART I

GOALS

Chapter 3

Semantics of Subgoals

This chapter is based on [van Riemsdijk et al., 2005c]. In order to be able to explain what we investigate in this chapter, we first need to make a distinction between two kinds of goals: *procedural* goals and *declarative* goals. Broadly speaking, a procedural goal is the goal to execute an action or sequence of actions, and a declarative goal is the goal to reach a certain state of affairs.[1] A declarative goal thus describes a desired situation.

Goals as introduced in Chapter 2 are generally viewed as declarative goals (see, e.g., [Hindriks et al., 2001, van Riemsdijk et al., 2005a]). The fact that these goals represent a desired situation, is used in the semantics by defining that a goal is removed from the goal base, once the agent believes the goal to be reached. It is generally accepted that an agent should at least be endowed with procedural goals. However, declarative goals also have a number of advantages, such as the added flexibility that they provide, as discussed in Section 2.3.2. We elaborate on advantages of declarative goals in Section 5.4.1.

We can now start explaining the issue investigated in this chapter. Goals as introduced in Chapter 2 are represented using a goal base (Definition 2.1). In this chapter, we are concerned with a different way of representing goals in an agent programming language.[2] That is, we are concerned with *subgoals*[3] as occurring often in the plans of an agent.

Plans are frequently built from basic actions which can be executed directly, and subgoals which can be viewed as representing a course of action in a more abstract way. An agent can for example have the plan to go to the bus stop, to take the bus into town,[4] and then to achieve the goal of buying a birthday

[1] We elaborate on the difference between procedural and declarative goals in Chapter 5.

[2] We refer to Chapter 5 for an overview of various approaches to the representation of goals in agent programming frameworks.

[3] A usage of the term subgoal that we do not consider in this chapter is usage in the logical sense, where for example p is considered to be a subgoal of the goal $p \wedge q$ [van Riemsdijk et al., 2005a].

[4] Assuming that both going to the bus stop and taking the bus into town are actions that

cake. This goal of buying a birthday cake will have to be fulfilled by selecting a more concrete plan of for example which shop to go to, etc.

Just as goals in general, subgoals of plans can also be categorized as either procedural or declarative. In the procedural interpretation, subgoals are linked directly to plans. Their only role is the abstract representation of a more concrete plan. In the declarative interpretation, the fact that the subgoal represents a desired state is somehow taken into account. In particular, the behavior of the agent depends in the case of declarative subgoals on whether the state represented by the subgoal is achieved (through the execution of a corresponding concrete plan, for example). In the birthday cake example, this means that it is important whether the execution of the concrete plan of which shop to go to etc., has resulted in a state in which the birthday cake is actually bought. If it turns out that the goal of buying the cake is not reached after having gone to the specific shop, the agent could select another plan to try a different shop, yielding more flexible agent behavior.

In this chapter, we study a cognitive agent programming language similar to the language of Chapter 2, without a goal base and plan selection rules. The language under consideration is essentially a propositional and otherwise somewhat simplified version of the first version of 3APL [Hindriks et al., 1999b], and we will in the rest of this chapter refer to our language simply as "3APL".

We consider it to be important to be able to express a declarative notion of subgoals in a cognitive agent programming language, and the aim of this chapter now is to investigate whether and if so, how, these declarative subgoals can be expressed in the language 3APL. In order to do this, we first make precise what we mean exactly by declarative subgoals, by defining a simple formal semantics for subgoals that interprets these in a declarative way (Sections 3.1 and 3.2). We then compare this semantics with the semantics of 3APL (Section 3.3). We argue that 3APL has a notion of subgoal, i.e., the abstract plans of 3APL can be viewed as subgoals, but this kind of subgoal is defined as a procedural kind of subgoal. It turns out, however, that although subgoals of 3APL are defined to have a procedural semantics, a 3APL agent can nevertheless be *programmed* to have these subgoals *behave* as declarative goals. This observation about 3APL (and a formal proof that it is correct) is the main contribution of this chapter.

3.1 Syntax

In this section and the next, we present the syntax and semantics of a simple programming language with plans containing subgoals that have a declarative interpretation. Throughout this chapter, and as in Chapter 2 (Definition 2.1), we assume a language of propositional logic $\mathcal{L}$ with negation and conjunction. We assume the language is based on a set of atoms Atom. In this chapter, we need to make the set of atoms on which $\mathcal{L}$ is based explicit, as it is used when

can be executed directly.

defining the language of plans (Definition 3.1). The symbol $\models$ will be used to denote the standard entailment relation for $\mathcal{L}$.

Below, we define the language of plans. This language of plans is similar to the plan language of Chapter 2 (Definition 2.3). That is, a plan is a sequence of basic actions and statements of the form $achieve(p)$ (subgoals), where $p \in \mathsf{Atom}$. Informally, basic actions can change the beliefs of an agent if executed, and a statement of the form $achieve(p)$ means that p should be achieved, before the agent can continue the execution of the rest of the plan.

Definition 3.1 *(plans)* Let $\mathsf{BasicAction}$ with typical element a be the set of basic actions and let $p \in \mathsf{Atom}$. The set of plans Plan with typical element π is then defined as follows.

$$\pi ::= a \mid achieve(p) \mid \pi_1; \pi_2$$

We use ϵ to denote the empty plan and identify $\epsilon; \pi$ and $\pi; \epsilon$ with π.

We use a simple plan language, focused on subgoals. The language could however be extended to include, e.g., test and non-deterministic choice. Also, the subgoals could be extended to arbitrary formulas, rather than just atoms. For atomic subgoals however, a correspondence with the procedural goals of 3APL can be established in a relatively simple way.

In order to be able to specify which plans can be used for achieving the subgoals, we use so-called plan generation rules. Informally, a plan generation rule $p \Rightarrow \pi$ specifies that the plan π can be selected to try to achieve the subgoal p. One could add a condition on the beliefs of the agent to the rule, specifying that the rule can only be applied if the agent has a certain belief. We however leave this out for reasons of simplicity.

Definition 3.2 *(plan generation rule)* The set of plan generation rules $\mathcal{R}_{\mathsf{PG}}$ is defined as follows: $\mathcal{R}_{\mathsf{PG}} = \{p \Rightarrow \pi \mid p \in \mathsf{Atom}, \pi \in \mathsf{Plan}\}$.

Plan generation rules can be compared with the plan selection rules of Chapter 2 (Definition 2.4) in the sense that both kinds of rules are used for specifying which plan can be used for achieving which (sub)goal. As will become clear in the sequel, however, their usage differs, since plan generation rules are used for selecting plans for subgoals as occurring in the plans of the agent, while plan selection rules are used for selecting plans for the goals in the goal base.

An agent in this chapter is a tuple, consisting of an initial belief base (a consistent set of formulas from $\mathcal{L}$ representing what the agent believes about the world), an initial plan, a set of plan generation rules and a belief update function $\mathcal{T}$, as also used in Chapter 2 (Definition 2.5).

Definition 3.3 *(subgoal achievement agent)* Let $\Sigma = \{\sigma \mid \sigma \subseteq \mathcal{L}, \sigma \not\models \bot\}$ be the set of belief bases. A subgoal achievement agent, typically denoted by $\mathcal{A}$, is

a tuple $\langle \sigma, \pi, \mathsf{PG}, \mathcal{T} \rangle$ where $\sigma \in \Sigma$ is the belief base, $\pi \in \mathsf{Plan}$ is the initial plan, and $\mathsf{PG} \subseteq \mathcal{R}_{\mathsf{PG}}$ is a set of plan generation rules. $\mathcal{T}$ is a partial function of type $(\mathsf{BasicAction} \times \Sigma) \rightarrow \Sigma$.

Configurations in this chapter consist of a belief base and a plan.

Definition 3.4 *(configuration)* A configuration is a pair $\langle \sigma, \pi \rangle$ where $\sigma \in \Sigma$ and $\pi \in \mathsf{Plan}$.

3.2 Semantics

In this section, we provide a semantics for the execution of plans containing subgoals. This semantics interprets these subgoals declaratively. We define the semantics using a transition system [Plotkin, 1981], as done in Chapter 2. Let $\mathcal{A} = \langle \sigma, \pi, \mathsf{PG}, \mathcal{T} \rangle$ be a subgoal achievement agent. The transition system $\mathsf{Trans}_{\mathcal{A}}$ for this agent is then given by the definitions below.

The transition rule for basic action execution is as in Chapter 2 (Definition 2.9), except that we do not need to take into account updates on the goal base here.

Definition 3.5 *(action execution)* Let $a \in \mathsf{BasicAction}$.

$$\frac{\mathcal{T}(a, \sigma) = \sigma'}{\langle \sigma, a; \pi \rangle \rightarrow \langle \sigma', \pi \rangle}$$

The following two definitions specify the possible transitions in case a statement of the form $achieve(p)$ is the first "action" of the plan. Both transitions rely upon a declarative interpretation of p, as it is checked whether p is believed to be reached. Definition 3.6 gives the transition in case p is achieved. The statement $achieve(p)$ is then removed from the plan.

Definition 3.6 *(subgoal achievement)*

$$\frac{\sigma \models p}{\langle \sigma, achieve(p); \pi \rangle \rightarrow \langle \sigma, \pi \rangle}$$

The next transition rule specifies the transition for an $achieve(p)$ statement in case p is not achieved. In this case, a plan should be generated in order to achieve p. This can be done if there is a plan generation rule of the form $p \Rightarrow \pi'$ in the rule base of the agent. The transition that can then be derived, specifies that the plan π' is placed at the head of the plan.

Definition 3.7 *(plan generation)* Let $p \Rightarrow \pi' \in \mathsf{PG}$.

$$\frac{\sigma \not\models p}{\langle \sigma, achieve(p); \pi \rangle \rightarrow \langle \sigma, \pi'; achieve(p); \pi \rangle}$$

It is important to note that the statement $achieve(p)$ is not removed from the plan if a plan generation rule is applied. If π' is executed and p is still not derivable from the agent's beliefs (p is not reached), a different rule with p as the head could be applied (if it exists), to achieve p by other means. In any case, a statement $achieve(p)$ will not be removed from the plan if p is not reached.

Given the transition system $\mathsf{Trans}_{\mathcal{A}}$ for subgoal achievement agent $\mathcal{A}$ as specified above, one can construct computation runs for $\mathcal{A}$. A computation run is a sequence of configurations, such that each consecutive configuration can be obtained from the previous through the application of a transition rule. The initial configuration of the computation run is formed by the initial belief base and plan of $\mathcal{A}$. A successful computation run is a run of which the final configuration has an empty plan. The semantics of $\mathcal{A}$ is then defined as the set of successful computation runs of $\mathcal{A}$.

Definition 3.8 *(semantics of a subgoal achievement agent)* Let $\mathcal{A} = \langle \sigma_0, \pi_0, \mathsf{PG}, \mathcal{T} \rangle$ be a subgoal achievement agent. Let a computation run be a sequence of configurations. A successful computation run of agent $\mathcal{A}$ is a computation run $\langle \sigma_0, \pi_0 \rangle, \ldots, \langle \sigma_n, \epsilon \rangle$, such that $\forall_{1 \leq i \leq n} : \langle \sigma_{i-1}, \pi_{i-1} \rangle \rightarrow \langle \sigma_i, \pi_i \rangle$ is a transition that can be derived in $\mathsf{Trans}_{\mathcal{A}}$. The semantics of $\mathcal{A}$ is the set $\{\theta \mid \theta \text{ is a successful computation run of } \mathcal{A}\}$.

A property of the semantics that reflects that subgoals are interpreted declaratively, is the following: if a plan of the form $achieve(p)$ is the initial plan of the agent, then it holds for any successful computation run of this agent ending in some belief base σ_n, that p follows from σ_n.

Proposition 3.1 Let $\mathcal{A} = \langle \sigma_0, \pi_0, \mathsf{PG}, \mathcal{T} \rangle$ be a subgoal achievement agent, and let $\theta = \langle \sigma_0, \pi_0 \rangle, \ldots, \langle \sigma_n, \epsilon \rangle$ be a successful computation run of $\mathcal{A}$. If π_0 is of the form $achieve(p)$, we have that $\sigma_n \models p$.

At this point we remark that the semantics of our *achieve* statement is closely related to the "bringing it about" operator as introduced in [Segerberg, 1989] in the area of philosophical logic. Segerberg's operator δ satisfies the property $[\delta p]p$ (expressed in a kind of dynamic logic), which would in our notation be the property $[achieve(p)]p$, stating that p always holds after the "execution" of $achieve(p)$. This is a reformulation of the above proposition in dynamic logic. A formal study of the relation of our work with that of Segerberg is left for future research.

3.3 Comparison with 3APL

3.3.1 Syntax and Semantics

In this section, we present the language 3APL as used in this chapter. A 3APL agent has a belief base (set of formulas from $\mathcal{L}$), a plan, a set of plan revision rules for manipulating its plan, and a belief update function $\mathcal{T}$.

The language of plans of 3APL agents is an extension of the plan language of Chapter 2 (Definition 2.3). It is also comparable with the plan language of Definition 3.1. A 3APL plan, however, does not contain *achieve* statements. The 3APL counterpart of these subgoals is the abstract plan. In the first paper on 3APL, abstract plans were called *achievement goals* [Hindriks et al., 1999b]. An abstract plan is basically a string, just as a basic action is a string (but as we will see, abstract plans have a different semantics). For the comparison with subgoal achievement agents however, we take the set of abstract plans as consisting not of an arbitrary set of strings, but of exactly the atoms of $\mathcal{L}$. Further, we add the possibility to test whether an atom follows from the belief base or not, and we add non-deterministic choice.

Definition 3.9 *(3APL plans)* Let $\mathsf{BasicAction}$ with typical element a be the set of basic actions and let $\mathsf{AbstractPlan}$ with typical element p be the set of abstract plans, such that $\mathsf{AbstractPlan} = \mathsf{Atom}$ and $\mathsf{AbstractPlan} \cap \mathsf{BasicAction} = \emptyset$. The set of 3APL plans Plan' with typical element π is then defined as follows.

$$\pi ::= a \mid p \mid p? \mid \neg p? \mid \pi_1; \pi_2 \mid \pi_1 + \pi_2$$

We use ϵ to denote the empty plan and identify $\epsilon; \pi$ and $\pi; \epsilon$ with π.

Abstract plans obtain their meaning through the plan revision rules of the 3APL agent. In this chapter, we, however, do not use the general kind of plan revision rules as introduced in Chapter 2 (Definition 2.4). That is, the head of a plan revision rule in this chapter consists of an abstract plan, rather than of an arbitrary composed plan. This restriction is needed in order to be able to compare 3APL agents with subgoal achievement agents. Also, plan revision rules do not have a belief condition.

Definition 3.10 *(plan revision rule)* The set of plan revision rules $\mathcal{R}_{\mathsf{PR}}$ is defined as follows: $\mathcal{R}_{\mathsf{PR}} = \{p \Rightarrow \pi \mid p \in \mathsf{AbstractPlan}, \pi \in \mathsf{Plan}'\}$.

Plan revision rules in this form thus very much resemble the plan generation rules of Definition 3.2 (syntactically, that is), but the body is a 3APL plan, i.e., a plan from Plan'. As we will explain shortly, the *semantics* of plan revision rules differs from that of plan generation rules in important ways.

The semantics of 3APL agents is defined by means of a transition system, as given below. The first transition rule is used to derive a transition for action execution, and is similar to the transition rule of this kind for subgoal achievement agents (Definition 3.5). The second transition specifies the application of a plan revision rule of the form $p \Rightarrow \pi'$ to a plan of the form $p; \pi$. If the rule is applied, the abstract plan p is replaced by the body of the rule, yielding the plan $\pi'; \pi$. It is important to note that it is *not* tested whether p holds, and further that p is *replaced* by π', rather than yielding the plan $\pi'; p; \pi$.

The transition rules for test and non-deterministic choice are fairly standard. Note, however, that a test for $\neg p$ succeeds if it is *not* the case that p follows

from the belief base, rather than having this test succeed if $\neg p$ *does* follow. The reason for this choice should become clear in the sequel. Further, some transitions are labeled with i, which we will also need in the sequel.

Definition 3.11 *(3APL transition system)* A 3APL agent $\mathcal{A}'$ is a tuple $\langle \sigma, \pi, \mathsf{PR}, \mathcal{T} \rangle$, where $\sigma \in \Sigma$, $\pi \in \mathsf{Plan}'$, $\mathsf{PR} \subseteq \mathcal{R}_{\mathsf{PR}}$ and $\mathcal{T}$ as in Definition 3.3. The transition system $\mathsf{Trans}_{\mathcal{A}'}$ for this 3APL agent is then defined as follows, where $a \in \mathsf{BasicAction}$ and $p \Rightarrow \pi' \in \mathsf{PR}$.

$$1)\ \frac{\mathcal{T}(a,\sigma) = \sigma'}{\langle \sigma, a; \pi \rangle \rightarrow \langle \sigma', \pi \rangle} \qquad 2)\ \frac{}{\langle \sigma, p; \pi \rangle \rightarrow_i \langle \sigma, \pi'; \pi \rangle}$$

$$3)\ \frac{\sigma \models p}{\langle \sigma, p?; \pi \rangle \rightarrow_i \langle \sigma, \pi \rangle} \qquad 4)\ \frac{\sigma \not\models p}{\langle \sigma, \neg p?; \pi \rangle \rightarrow_i \langle \sigma, \pi \rangle}$$

$$5)\ \frac{\langle \sigma, \pi_1 \rangle \rightarrow \langle \sigma', \pi_1' \rangle}{\langle \sigma, (\pi_1 + \pi_2); \pi \rangle \rightarrow \langle \sigma', \pi_1'; \pi \rangle} \qquad 6)\ \frac{\langle \sigma, \pi_2 \rangle \rightarrow \langle \sigma', \pi_2' \rangle}{\langle \sigma, (\pi_1 + \pi_2); \pi \rangle \rightarrow \langle \sigma', \pi_2'; \pi \rangle}$$

Before we move on to formally investigating the relation between 3APL and subgoal achievement agents, we elaborate on the notion of an abstract plan or achievement goal as used in 3APL. Hindriks et al. remark the following with respect to achievement goals:

> *Achievement goals are atomic propositions from the logical language $\mathcal{L}$. The use of atoms as achievement goals, however, 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 serve as an abstraction mechanism like procedures in imperative programming and have a procedural meaning.* [Hindriks et al., 1999b, page 363]

Hindriks et al. thus take the set of achievement goals/abstract plans to be the atoms from $\mathcal{L}$ (a first order language in their case). Then they remark that although achievement goals are atoms, they do *not* have a declarative interpretation. The fact that an achievement goal is an atom and could thus in principle be tested for example against the belief base, is not used in defining its semantics. The language of achievement goals could thus have been any language of strings (which is in fact the approach of later papers [van Riemsdijk et al., 2003b, Dastani et al., 2004]). Hindriks et al. however do remark the following with respect to a possible assertional reading of achievement goals:[5]

[5] As a first order language is used in [Hindriks et al., 1999b], the original text states $p(\vec{t})$ instead of p. This is a predicate name parameterized with a sequence of terms.

> *Apart from the procedural reading of these goals, however, an assertional reading is also possible. An achievement goal p would then be interpreted as specifying a goal to achieve a state of affairs such that p. We think such a reading is valid in case the plans for achieving an achievement goal p actually do establish p.* [Hindriks et al., 1999b, page 363]

The *"plans for achieving an achievement goal p"* are the plans as specified through the plan revision rules, i.e., a plan revision rule $p \Rightarrow \pi$ specifies that π is a plan for achieving p. According to Hindriks et al., this assertional or declarative reading of achievement goals is thus *only* valid under the strong requirement that π actually reaches p. This is thus in contrast with the semantics for subgoals as we have introduced, as these subgoals are by definition interpreted in a declarative manner.

3.3.2 3APL and Subgoals

We will show in this section that, although the semantics of plan generation rules and plan revision rules differ in important ways, it *is* possible to define a mapping from an arbitrary subgoal achievement agent to a 3APL agent, such that the 3APL agent "simulates" the behavior of the subgoal achievement agent. In the sequel, the plan π_s denotes π in which all occurrences of statement of the form $achieve(p)$ are replaced with p.

We first remark that the naive translation, in which a plan generation rule $p \Rightarrow \pi$ is translated to a plan revision rule $p \Rightarrow \pi_s$, does not do the trick. The reason that this translation does not work, is precisely the difference of interpretation between *achieve* statements and abstract plans, i.e., declarative versus procedural. If an abstract plan p occurs at the head of a plan $p;\pi$, the plan revision rule $p \Rightarrow \pi'$ can be applied, yielding $\pi';\pi$. After the execution of π', the plan π will be executed, *regardless* of whether p is actually achieved at that point. In the case of a plan $achieve(p);\pi$ of a subgoal achievement agent, the plan generation rule can be applied (if p is not believed), yielding the plan $\pi';achieve(p);\pi$. After the execution of π', the agent will test whether p is achieved. If it is achieved, it will continue with the execution of π. If however p is *not* achieved, it will apply a rule once more to generate a plan to achieve p. It is nevertheless important to mention that *if* it is the case that π' actually establishes p (and this holds for all plan generation rules), it *can* be proven that the 3APL agent as obtained through this naive translation, simulates the subgoal achievement agent. For reasons of space, we however omit this proof.

Translation

We now turn to the translation of a subgoal achievement agent into a 3APL agent, for which it holds that the 3APL agent as obtained in this way, simulates

the subgoal achievement agent. As would be expected, the important part of the translation is the mapping of plan generation rules onto plan revision rules.

Definition 3.12 *(transformation of subgoal achievement agent into 3APL agent)* Let $s : \mathsf{Plan} \to \mathsf{Plan}'$ be a function that takes a plan π of a subgoal achievement agent (Definition 3.1), and yields this plan in which all statements of the form $achieve(p)$ are replaced by p, thus yielding a plan in Plan' (Definition 3.9). We will in the sequel use the notation π_s for $s(\pi)$.

The function $t : \mathcal{R}_{\mathsf{PG}} \to \mathcal{R}_{\mathsf{PR}}$, taking a plan generation rule and yielding a corresponding plan revision rule, is then defined as follows.

$$t(p \Rightarrow \pi) \;=\; p \Rightarrow ((\neg p?; \pi_s; p) + p?)$$

The function t is lifted to sets of plan generation rules in the obvious way.

Let $\mathcal{A} = \langle \sigma, \pi, \mathsf{PG}, \mathcal{T} \rangle$ be a subgoal achievement agent. The 3APL agent corresponding with $\mathcal{A}$ is then $\langle \sigma, \pi_s, t(\mathsf{PG}), \mathcal{T} \rangle$. Finally, we define a function τ that takes a configuration from the transition system of $\mathcal{A}$ of the form $\langle \sigma, \pi \rangle$, and yields the configuration $\langle \sigma, \pi_s \rangle$.

Informally, this mapping can be used to obtain a 3APL agent that simulates a subgoal achievement agent, because of the following. Consider a 3APL agent with the plan $p; \pi$, and the plan revision rule $p \Rightarrow ((\neg p?; \pi'_s; p) + p?)$ as obtained from the plan generation rule $p \Rightarrow \pi'$. This plan revision rule can then be applied to this plan (regardless of whether p is believed or not), yielding the plan $((\neg p?; \pi'_s; p) + p?); \pi$.

Now assume that p is believed. In that case, the test $p?$ succeeds and $\neg p?$ fails, which means that the plan in the next configuration will have to be π. This thus implements that p is skipped if believed to be achieved, which corresponds with the semantics of the statement $achieve(p)$.

Now assume that p is not believed. In that case, the plan in the next configuration will have to be $\pi'_s; p; \pi$. This corresponds with the semantics of $achieve(p)$ in case p is not believed: the plan π' is placed at the head of the plan, not replacing the $achieve(p)$ statement. After the execution of π'_s, we are left with the plan $p; \pi$. The plan revision rule $p \Rightarrow ((\neg p?; \pi'_s; p) + p?)$ (or a different rule with p as the head) will then be applied again. If p is achieved, the agent will continue with the execution of π as explained. If p is not achieved, the mechanism as just described will be set in motion. All this thus corresponds with the behavior of *achieve* statements in the subgoal achievement agent.

Bisimulation Theorem

We now move on to formally establishing this correspondence. For this, we introduce the notion of a translation bisimulation as used in [Hindriks, 2001, Chapter 8] (slightly adapted). Informally, a translation bisimulation translates an agent from a so-called source language to an agent from the target language

that "can do the same things". In our case, we translate subgoal achievement agents to 3APL agents.

We have to show that for each transition in the transition system for a subgoal achievement agent $\mathcal{A}$, there is a corresponding transition in the transition system for the corresponding 3APL agent $\mathcal{A}'$. This "transition" in $\mathsf{Trans}_{\mathcal{A}'}$, actually does not have to be a single transition, but may consist of a number of so-called idle transitions, and one non-idle transition. The idle transitions in $\mathsf{Trans}_{\mathcal{A}'}$ are those labelled with i. Intuitively, these idle transitions form implementation details of $\mathcal{A}'$, and do not have to be matched by a transition of $\mathcal{A}$.[6] In the sequel, the transition relations of $\mathcal{A}$ and $\mathcal{A}'$ will respectively be denoted by $\rightarrow^{\mathcal{A}}$ and $\rightarrow^{\mathcal{A}'}$, and $\rightarrow_i^{\mathcal{A}'}$ denotes the restriction of $\mathcal{A}'$ to idle transitions.

The new transition relation that abstracts from idle steps is denoted by $\rightarrow_*^{\mathcal{A}'}$. It only exists for $\mathsf{Trans}_{\mathcal{A}'}$, as $\mathsf{Trans}_{\mathcal{A}}$ does not contain idle steps. It is defined as follows, where d_j with $1 \leq j \leq n$ are configurations derivable in $\mathsf{Trans}_{\mathcal{A}'}$: $d_1 \rightarrow_*^{\mathcal{A}'} d_n$ iff there is a (possibly empty) series of idle transitions $d_1 \rightarrow_i^{\mathcal{A}'} d_2 \rightarrow_i^{\mathcal{A}'} \ldots \rightarrow_i^{\mathcal{A}'} d_{n-1}$ and a single non-idle transition $d_{n-1} \rightarrow^{\mathcal{A}'} d_n$.

If we can show that for each transition in the transition system for a subgoal achievement agent $\mathcal{A}$, there is a corresponding transition in the transition system for the corresponding 3APL agent $\mathcal{A}'$, we will have established that $\mathcal{A}'$ generates *at least* the behavior of $\mathcal{A}$. In order to establish that $\mathcal{A}'$ does not generate any (alternative) behavior not having a counterpart in $\mathcal{A}$, we also have to show that any non-idle transition of $\mathcal{A}'$ corresponds with a transition of $\mathcal{A}$. A transition $c_1 \rightarrow^{\mathcal{A}} c_2$ corresponds with a transition $d_1 \rightarrow^{\mathcal{A}'} d_2$ or $d_1 \rightarrow^* d_2$ iff $d_1 = \tau(c_1)$ and $d_2 = \tau(c_2)$.

The result can only be proven if we assume that at least one plan generation rule of the form $p \Rightarrow \pi$ exists for every $p \in \mathsf{Atom}$. If this would not be the case, there would be a mismatch: a statement $achieve(p)$ could be removed from a plan if p holds (without there being a plan generation rule for p), but an abstract plan p can only be "removed" if first a plan revision rule is applied.

Theorem 3.1 *(translation bisimulation)* Let $\mathcal{A} = \langle \sigma, \pi, \mathsf{PG}, \mathcal{T} \rangle$ be a subgoal achievement agent such that for each $p \in \mathsf{Atom}$ there is at least one rule of the form $p \Rightarrow \pi$ in PG, and let $\mathcal{A}' = \langle \sigma, \pi_s, t(\mathsf{PG}), \mathcal{T} \rangle$ be the corresponding 3APL agent. We then have that for every configuration c_1 of $\mathcal{A}$, $d_1 = \tau(c_1)$ implies the following:

1. If $c_1 \rightarrow^{\mathcal{A}} c_2$, then $d_1 \rightarrow_*^{\mathcal{A}'} d_2$, such that $d_2 = \tau(c_2)$.
2. If $d_1 \rightarrow^{\mathcal{A}'} d_2$, then for some c_2, $c_1 \rightarrow^{\mathcal{A}} c_2$, such that $d_2 = \tau(c_2)$.

[6] The choice of idle transitions for 3APL might seem strange, as the application of a plan revision rule is an idle transition, whereas the non-deterministic choice is not, although the latter might seem an implementation detail, rather than the former. The reason is, that the application of a plan revision rule cannot be matched directly with a transition of a goal achievement agent, whereas the particular usage of non-deterministic choice, as specified through the translation, *can*.

Proof: 1. We have to show that for every transition $c_1 \rightarrow^{\mathcal{A}} c_2$ in $\mathsf{Trans}_{\mathcal{A}}$, there is a corresponding (sequence of) transition(s) $d_1 \rightarrow_*^{\mathcal{A}'} d_2$ in $\mathsf{Trans}_{\mathcal{A}'}$ such that $d_2 = \tau(c_2)$.

Let $\langle \sigma, a; \pi \rangle \rightarrow^{\mathcal{A}} \langle \sigma', \pi \rangle$ be a transition as derived through the transition rule of Definition 3.5. We then have that the transition $\langle \sigma, a; \pi_s \rangle \rightarrow^{\mathcal{A}'} \langle \sigma', \pi_s \rangle$ can be derived in $\mathsf{Trans}_{\mathcal{A}'}$, by means of the first transition rule. We also have that $\langle \sigma', \pi_s \rangle = \tau(\langle \sigma', \pi \rangle)$, yielding the desired result for action execution transitions.

Let $\langle \sigma, achieve(p); \pi \rangle \rightarrow^{\mathcal{A}} \langle \sigma, \pi \rangle$ be a transition as derived through the transition rule of Definition 3.6, which means that $\sigma \models p$ has to hold. Let $p \Rightarrow \pi'$ be a plan generation rule of $\mathcal{A}$ (a rule of this form has to exist by assumption). We then have, because $t(p \Rightarrow \pi')$ is a plan revision rule of $\mathcal{A}'$, that the transitions

$$\langle \sigma, p; \pi_s \rangle \rightarrow_i^{\mathcal{A}'} \langle \sigma, ((\neg p?; \pi'_s; p) + p?); \pi_s) \rangle \rightarrow^{\mathcal{A}'} \langle \sigma, \pi_s \rangle$$

can be derived in $\mathsf{Trans}_{\mathcal{A}'}$, by means of the transition rule for plan revision and those for test and non-deterministic choice. We also have that $\langle \sigma, \pi_s \rangle = \tau(\langle \sigma, \pi \rangle)$, yielding the desired result for subgoal achievement transitions.

Let $\langle \sigma, achieve(p); \pi \rangle \rightarrow^{\mathcal{A}} \langle \sigma, \pi'; achieve(p); \pi \rangle$ be a transition as derived through the transition rule of Definition 3.7, which means that $\sigma \not\models p$ has to hold, and $p \Rightarrow \pi'$ has to be a plan generation rule in PG. We then have that the transitions

$$\langle \sigma, p; \pi_s \rangle \rightarrow_i^{\mathcal{A}'} \langle \sigma, ((\neg p?; \pi'_s; p) + p?); \pi_s) \rangle \rightarrow^{\mathcal{A}'} \langle \sigma, \pi'_s; p; \pi_s \rangle$$

can be derived in $\mathsf{Trans}_{\mathcal{A}'}$, by means of the transition rule for plan revision and those for test and non-deterministic choice. We also have that $\langle \sigma, \pi'_s; p; \pi_s \rangle = \tau(\langle \sigma, \pi'; achieve(p); \pi \rangle)$, yielding the desired result for plan generation transitions. We have shown the desired result for every transition $c_1 \rightarrow^{\mathcal{A}} c_2$, thereby proving 1.

2. We have to show that for every non-idle transition $d_1 \rightarrow^{\mathcal{A}'} d_2$ with $d_1 = \tau(c_1)$ for some c_1 in $\mathsf{Trans}_{\mathcal{A}'}$, there is a corresponding transition $c_1 \rightarrow^{\mathcal{A}} c_2$ in $\mathsf{Trans}_{\mathcal{A}}$ such that $d_2 = \tau(c_2)$.

The only configurations d_1 of $\mathcal{A}'$ for which it holds that there is a c_1 such that $d_2 = \tau(c_1)$, are configurations of the form $\langle \sigma, a; \pi \rangle$ or $\langle \sigma, p; \pi \rangle$, where π does not contain tests or non-deterministic choice. We thus have to show that if there is a non-idle transition from one of these configurations to another, that there is a matching transition in $\mathcal{A}$. The case for the action execution transition (with $d_1 = \langle \sigma, a; \pi \rangle$) is analogous to the proof as given in part 1. As for the configuration of the form $\langle \sigma, p; \pi \rangle$, we have that there is no possible non-idle transition from this configuration. The result thus follows immediately.

We now have to prove the result for the non-deterministic choice. In general, this cannot be proven, as there is no non-deterministic choice in subgoal achievement agents. From the transformation of $\mathcal{A}$ into $\mathcal{A}'$, we however know that the non-deterministic choice operator can only occur in plan revision rules in a very

specific format: the plans of $\mathcal{A}$ (the initial plan and the bodies of plan generation rules) do not contain the non-deterministic choice operator, and therefore the plans of $\mathcal{A}'$ do neither (except for the non-deterministic choice in plan revision rules, as introduced through the translation). We thus have to prove the result for tuples of the form $\langle \sigma, ((\neg p?; \pi'; p) + p?); \pi \rangle$, where $p \Rightarrow ((\neg p?; \pi'; p) + p?)$ is a plan revision rule of $\mathcal{A}'$.

Let $\langle \sigma, ((\neg p?; \pi'; p) + p?); \pi \rangle \rightarrow \langle \sigma, \pi \rangle$ be a transition as derived through the second transition rule for non-deterministic choice of Definition 3.11, which means that $\sigma \models p$ has to hold. We have that π' and π do not contain tests of the non-deterministic choice operator. Let the function s' be the inverse of s, i.e., taking a plan from Plan' without test and non-deterministic choice, and yielding a plan from Plan, by replacing all occurrences of abstract plans of the form p with $achieve(p)$. We then have that $\langle \sigma, achieve(p); \pi_{s'} \rangle \rightarrow \langle \sigma, \pi_{s'} \rangle$ is a transition of $\mathcal{A}$. We also have that $\langle \sigma, \pi \rangle = \tau(\langle \sigma, \pi_{s'} \rangle)$, yielding the desired result in case $\sigma \models p$.

The case for $\sigma \not\models p$ is analogous. Let $\langle \sigma, ((\neg p?; \pi'; p) + p?); \pi \rangle \rightarrow \langle \sigma, \pi'; p; \pi \rangle$ be a transition as derived through the first transition rule for non-deterministic choice of Definition 3.11, which means that $\sigma \not\models p$ has to hold. We then have that $\langle \sigma, achieve(p); \pi_{s'} \rangle \rightarrow \langle \sigma, \pi'_{s'}; achieve(p); \pi_{s'} \rangle$ is a transition of $\mathcal{A}$ (as $p \Rightarrow ((\neg p?; \pi'; p) + p?)$ is a plan revision rule of $\mathcal{A}'$, which means that $p \Rightarrow \pi'_{s'}$ will have to be a plan generation rule of $\mathcal{A}$). We also have that $\langle \sigma, \pi'; p; \pi \rangle = \tau(\langle \sigma, \pi'_{s'}; achieve(p); \pi_{s'} \rangle)$, yielding the desired result in case $\sigma \not\models p$.

Covering all cases, we have proven 2. □

3.4 Conclusion and Related Work

In this chapter, we have studied the relation between declarative and procedural interpretations of subgoals as occurring in the plans of cognitive agents. In particular, we have compared our definition of declaratively interpreted subgoals with the semantics of the procedurally interpreted achievement goals in the language 3APL. As we have shown, it is possible to obtain a 3APL agent that simulates the behavior of the subgoal achievement agent, by translating plan generation rules to plan revision rules in a specific way.

The 3APL family of languages [Hindriks et al., 1999b] [van Riemsdijk et al., 2003b, Dastani et al., 2004] is an example of a set of languages in which subgoals are interpreted procedurally. Languages and platforms from the AgentSpeak family [Ingrand et al., 1992, d'Inverno et al., 1998, Rao, 1996, Moreira and Bordini, 2002, Evertsz et al., 2004] also have a procedural view on subgoals, although the mechanism differs from that of 3APL (see also [Hindriks et al., 1998] for an embedding of AgentSpeak(L) in 3APL).

Research similar to that described in this chapter has been done in the context of Jason [Hübner et al., 2006]. Jason [Bordini et al., 2005b] is an implementation of an interpreter for an extended version of AgentSpeak(L). In

[Hübner et al., 2006], several transformations on the plans of Jason agents are proposed, such that the subgoals of the resulting agent behave as declarative goals. However, it is, in contrast with this chapter, not shown formally that the resulting agent behaves according to a certain specification.

To the best of our knowledge, this is the first time that a correspondence between declarative and procedural subgoals is investigated and established formally. We believe that the investigations as described in this chapter shed some light on the expressiveness of languages with procedural goals, and that this is one piece of the puzzle of the incorporation of declarative goals in cognitive agent programming languages.

Chapter 4

Goals in Conflict

This chapter is based on [van Riemsdijk et al., 2005b]. The title of this chapter was inspired by the title of the PhD thesis of Harrenstein: *Logic in Conflict: Logical Explorations in Strategic Equilibrium* [Harrenstein, 2004].

In Chapter 3, we have investigated semantics of subgoals as occurring in the plans of agents. In this chapter, we focus on representations of goals that are not directly linked to the representation of plans. An example of such a representation is the goal base as used in Chapter 2 (Definition 2.1). In this chapter, we investigate the representation of goals using a goal base, and propose a new kind of representation. Based on such representations, we investigate how the semantics of goal formulas can be defined. One such semantics of goal formulas was proposed in Chapter 2 (Definition 2.7).

The focus of the researches in this chapter will be on the representation of *conflicting* goals. As others have argued [Hindriks et al., 2001], we maintain that it is natural for an agent to have conflicting objectives. As will become clear in the sequel, the representation of conflicting goals gives rise to interesting issues when it comes to defining the semantics of goal formulas.

As an addition to representing goals by means of a goal base, we propose in this chapter a representation by means of so-called *goal adoption rules*. These goal adoption rules can be used to specify that goals can be conditional on beliefs and other goals. The definition of the semantics of goal formulas on the basis of goal adoption rules makes use of default logic. In order to get a better understanding of the semantics we propose, we establish relations between these semantics and investigate their properties.

One of the reasons for wanting to specify what an agent's goals are, is that the agent should generate plans on the basis of its goals, in order to achieve them. In this chapter, the goal for which a plan was generated is recorded with the plan. These structures are termed *intentions* here. Although it is not the main focus of this chapter, we discuss ways of generating these intentions, as intention generation is closely related to the specification of an agent's goals.

This chapter is organized as follows. First, we present some preliminaries (Section 4.1). Section 4.2 discusses semantics of goals, based on the goal base of the agent. In Section 4.3, we propose a semantics of goals based on the goal base and the goal adoption rules of the agent, and investigate properties of this semantics. In Section 4.4, we address intention generation and other issues related to the dynamic behavior of the agent. Section 4.5 discusses related work. In particular, we discuss the relation between our work and the work of Horty [Horty, 1993, Horty, 1994, Horty, 1997] in the context of deontic logic in detail. As it turns out, his work is closely related to our work. Concluding, we remark that although consideration of issues of computational complexity is important, this is not addressed in this chapter and remains for future research.

4.1 Preliminaries

4.1.1 Cognitive Agent Programming

In this section, we present the cognitive agent programming framework that we take as the starting point for our investigations in this chapter. Throughout this chapter, we assume a language of propositional logic $\mathcal{L}$ with negation and conjunction, with typical element ϕ. $\top \in \mathcal{L}$ will be used to denote a tautology, $\bot \in \mathcal{L}$ to denote falsum and $\models$ will be used to denote the standard entailment relation for $\mathcal{L}$. Further, we assume a language of plans Plan with typical element π. This could for example be the language of plans of Chapter 2 (Definition 2.3). An exact specification is not needed for the purpose of this chapter and therefore we will not provide one.

Agent configurations in this chapter consist of a belief base, a goal base, an intention base, and a rule base. The intention base is a set of pairs from $\mathsf{Plan} \times \mathcal{L}$. The idea is, that a pair $\langle \pi, \phi \rangle$ represents a selected plan with an associated goal that the plan is to achieve (see also [Dastani et al., 2004]). At this point, we are not yet specific about which types of rules constitute the rule base. We will gradually define this component in the sequel.

Definition 4.1 *(agent configuration)* An agent configuration, typically denoted by c, is a tuple $\langle \sigma, \gamma, \iota, \mathcal{R} \rangle$ where $\sigma \subseteq \mathcal{L}$ is the belief base, $\gamma \subseteq \mathcal{L}$ is the goal base, $\iota \subseteq (\mathsf{Plan} \times \mathcal{L})$ is the intention base and $\mathcal{R}$ is a tuple of sets of rules. All sets σ, γ, ι and sets in $\mathcal{R}$ are finite.

In this chapter, configurations thus contain a *set* of intentions, rather than a *single* plan, as in the rest of this thesis. Considering a set of intentions or plans is common in many agent programming frameworks (see, e.g., [Rao, 1996, Hindriks et al., 1999b, Dastani et al., 2004, Pokahr et al., 2005b]). In this chapter, it is especially interesting to consider a set of intentions, since this allows us to investigate issues related to conflicts among intentions. In contrast with other chapters in this thesis, we incorporate the rule base in configurations in

this chapter. This is done because the semantics of goal formulas to be introduced in Section 4.3 is defined using the rule base. Incorporating the rule base in configurations will increase the readability of the definitions.

Belief and goal formulas are as defined in Chapter 2 (Definition 2.2). We repeat that definition here.

Definition 4.2 *(belief and goal formulas)* The belief formulas $\mathcal{L}_{\mathsf{B}}$ with typical element β and the goal formulas $\mathcal{L}_{\mathsf{G}}$ with typical element κ are defined as follows, where $\phi \in \mathcal{L}$.

$$\beta ::= \top \mid \mathbf{B}\phi \mid \neg\beta \mid \beta_1 \wedge \beta_2$$
$$\kappa ::= \top \mid \mathbf{G}\phi \mid \neg\kappa \mid \kappa_1 \wedge \kappa_2$$

The semantics of belief formulas is also as in Chapter 2 (Definition 2.7), except that the structure of the configurations on the basis of which the semantics is defined, differs.

Definition 4.3 *(semantics of belief formulas)* Let $\phi \in \mathcal{L}$ and let $\langle \sigma, \gamma, \iota, \mathcal{R} \rangle$ be an agent configuration. Let $\beta \in \mathcal{L}_{\mathsf{B}}$. The semantics $\models_{\mathcal{L}_{\mathsf{B}}}$ of belief formulas is then as defined below.

$$\begin{array}{lll}
\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_{\mathcal{L}_{\mathsf{B}}} \top & & \\
\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_{\mathcal{L}_{\mathsf{B}}} \mathbf{B}\phi & \Leftrightarrow & \sigma \models \phi \\
\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_{\mathcal{L}_{\mathsf{B}}} \neg\beta & \Leftrightarrow & \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \not\models_{\mathcal{L}_{\mathsf{B}}} \beta \\
\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_{\mathcal{L}_{\mathsf{B}}} \beta_1 \wedge \beta_2 & \Leftrightarrow & \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_{\mathcal{L}_{\mathsf{B}}} \beta_1 \text{ and } \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_{\mathcal{L}_{\mathsf{B}}} \beta_2
\end{array}$$

Investigating ways to define the semantics of goal formulas is the main research objective of this chapter and these analyses will be carried out in Sections 4.2 and 4.3.

Although the focus of this chapter is on the semantics of goals and we do not strive to provide a complete agent programming framework, we do present a way to generate intentions on the basis of certain beliefs and goals. In an agent programming setting, it is common to introduce rules to generate plans or intentions. In Chapter 2, for example, this was done using plan selection rules (Definition 2.4). We will elaborate on why it is interesting to consider intention generation rules in this chapter after defining their semantics.

Definition 4.4 *(intention generation rule)* The set of intention generation rules $\mathcal{R}_{\mathsf{IG}}$ is defined as follows:

$$\{\beta, \kappa \Rightarrow_{\mathbf{I}} \langle \pi, \phi \rangle \mid \beta \in \mathcal{L}_{\mathsf{B}}, \kappa \in \mathcal{L}_{\mathsf{G}}, \pi \in \mathsf{Plan}, \phi \in \mathcal{L}\}.$$

An intention generation rule $\beta, \kappa \Rightarrow_{\mathbf{I}} \langle \pi, \phi \rangle$ can be applied in a configuration, if the belief and goal conditions β and κ of the rule hold in that configuration. If

this rule is applied, the intention $\langle \pi, \phi \rangle$ consisting of a plan π and a goal ϕ that is to be achieved by the plan, is added to the existing set of intentions.[1]

The applicability of an intention generation rule, however, not only depends on the belief and goal conditions of the rule. That is, the existing intentions should be taken into account, since newly adopted intentions should not conflict with already existing ones. An intention generation rule can only be applied if the intention in its consequent is not conflicting with already existing ones. This idea was put forward by the philosopher Bratman [Bratman, 1987], who used the term "screen of admissibility" for this role played by existing intentions. It can be incorporated into our framework by requiring that the goal of the intention that is to be adopted is consistent with goals of already existing intentions.[2]

These ideas are formalized below in the definition of the semantics of application of an intention generation rule. The semantics is defined by giving a transition rule [Plotkin, 1981].

Definition 4.5 *(semantics of intention generation)* Let $\mathsf{IG} \subseteq \mathcal{R}_{\mathsf{IG}}$ be a finite set of intention generation rules, let $\mathcal{R} = \langle \mathsf{IG} \rangle$ and let $\beta, \kappa \Rightarrow_{\mathbf{I}} \langle \pi, \phi \rangle \in \mathsf{IG}$ be an intention generation rule. Further, let $c = \langle \sigma, \gamma, \iota, \mathcal{R} \rangle$ be an agent configuration and let $\models_{\mathcal{L}_{\mathsf{G}}}$ be a satisfiability relation for goal formulas. The semantics of applying this rule is then as follows, where $\iota' = \iota \cup \{\langle \pi, \phi \rangle\}$ and $\delta = \{\phi_\iota \mid \langle \pi_\iota, \phi_\iota \rangle \in \iota\}$.

$$\frac{c \models_{\mathcal{L}_{\mathsf{B}}} \beta \qquad c \models_{\mathcal{L}_{\mathsf{G}}} \kappa \qquad \{\phi\} \cup \delta \not\models \bot}{\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \rightarrow \langle \sigma, \gamma, \iota', \mathcal{R} \rangle}$$

Note that an intention generation rule of the form $\beta, \kappa \Rightarrow_{\mathbf{I}} \langle \pi, \phi \rangle$ for which ϕ is inconsistent, i.e., $\phi \models \bot$, will never be applicable.

We introduce intention generation rules in this chapter for three reasons. First, intention generation rules will be used to partly explain the introduction of a new semantics for goals in Section 4.2.1. The applicability of an intention generation rule *depends* on the truth of the goal formula in its antecedent and different choices in defining the semantics for goal formulas will thus influence the applicability of the rule. Second, we will argue in Section 4.2.1 that the goal base of an agent does not have to be consistent, partly because agents in

[1] The formula κ denotes a condition on the goals of the agent, which should be satisfied for the rule to be applicable, and ϕ denotes the goal which the plan π should achieve. Generally, these rules will be of the form $\beta, \mathbf{G}\phi \Rightarrow_{\mathbf{I}} \langle \pi, \phi \rangle$, in which case we could automatically add ϕ to the intention, instead of having the programmer specify the intention as a pair $\langle \pi, \phi \rangle$. We however also allow composed goal formulas such as $\mathbf{G}\phi \wedge \neg \mathbf{G}\phi'$ as the goal condition of intention generation rules, in which case it is less clear what the goal of the plan π should be. Therefore, the programmer has to specify which goal the plan is to achieve.

[2] We do not claim that intentions in our framework capture all aspects of intention as put forward by Bratman, and as later formalized in, e.g., [Cohen and Levesque, 1990, Rao and Georgeff, 1991]. Nevertheless, since intentions in our proposal provide some form of screen of admissibility, which is an important aspect of Bratman's theory, we do feel it is justified to use the term "intentions" here. Also, other agent programming frameworks often use this term for the plans of the agent (see, e.g., [Rao, 1996]).

our framework have an intention base that *is* consistent. If the initial intention base of the agent is consistent, our semantics of intention generation rules will maintain this property throughout the execution of the agent. Third, rules such as intention generation rules are commonly used in agent programming languages. As it turns out, there are interesting aspects to the relation between the semantics of goals we propose in this chapter, and the process of intention generation through the use of intention generation rules. Although intention generation is not the main focus of this chapter, we will elaborate on this in Section 4.4.2.

Concluding, we make the following remark. The check of whether the new intention is not conflicting with existing intentions, is implemented by checking logical consistency of the goal of the new intention, with the goals of existing ones. In general, one could consider other conditions for a new intention to be compatible with existing ones, such as conditions on resources like energy or money that are used by the plans (see, for example, [Thangarajah et al., 2002] for a more elaborate treatment of this topic) or conditions on subgoals that should be reached by the plans. Investigating more elaborate definitions of intention compatibility is however not within the scope of this chapter and the simple definition as given above will suffice for our purposes.

4.1.2 Default Logic

In Section 4.3, we will use default logic [Reiter, 1980] to define the semantics of declarative goals. In this section, we briefly sketch the ideas of default logic. For more elaborate treatments of this topic, the reader can for example consult [Antoniou, 1997, Brewka et al., 1997]. Default logic is generally based on predicate logic, but for this chapter it suffices to consider propositional default logic.

Default logic distinguishes facts, representing certain but incomplete information about the world, and default rules or defaults, representing rules of thumb, by means of which conclusions can be drawn that are plausible, but not necessarily true. This means that some conclusions may have to be revised when more information becomes available. Given the propositional language $\mathcal{L}$ which we introduced in Section 4.1.1, a *default rule* has the form $\phi : \psi_1, \ldots, \psi_n / \chi$, where $\phi, \psi_1, \ldots, \psi_n, \chi \in \mathcal{L}$ and $n > 0$. The intuitive reading of a default rule of this form is the following: if ϕ is provable and for all $1 \leq i \leq n$, $\neg\psi_i$ is not provable, i.e., if it is consistent to assume ψ_i, then derive χ. The formula ϕ is called the prerequisite and the formulas $\psi_1, \ldots, \psi_n$ are called the justifications of the default rule.

A *default theory* [Brewka et al., 1997] is a pair $\langle W, D \rangle$, where $W \subseteq \mathcal{L}$ is the set of facts and D is a set of default rules. The semantics of a default theory $\langle W, D \rangle$ can be defined through so-called *extensions* of the theory. If $E \subseteq \mathcal{L}$ is a set of propositional formulas, then a sequence of sets of formulas $E_0, E_1, \ldots$ is defined as follows, where $\models$ is the standard entailment relation for $\mathcal{L}$ and

$Th(E_i)$ is the closure under classical logical consequence of E_i.

$$\begin{array}{lcl} E_0 & = & W \\ E_{i+1} & = & Th(E_i) \cup \{\chi \mid \phi : \psi_1, \ldots, \psi_n/\chi \in D,\ E_i \models \phi,\ E \not\models \neg\psi_i\} \end{array}$$

A set $E \subseteq \mathcal{L}$ is then an extension of $\langle W, D\rangle$ iff $E = \bigcup_{i=0}^{\infty} E_i$. In the sequel, we will sometimes be somewhat sloppy and say that, e.g., $\{p\}$ is an extension, where we should, strictly speaking, say that $Th(\{p\})$ is an extension.

It is important to note that extensions are always *consistent* sets[3] that are *closed* under the application of default rules. A rule $\phi : \psi_1, \ldots, \psi_n/\chi$ is *applicable* to an extension E iff $E \models \phi$ and $E \not\models \neg\psi_i$ for $1 \leq i \leq n$. An extension E of a default theory $\langle W, D\rangle$ is closed under the application of default rules, iff it holds for all rules $\phi : \psi_1, \ldots, \psi_n/\chi \in D$, that if the rule is applicable to E, then $E \models \chi$.

Example 4.1 Let $W = \{a\}$, let $d_1 = a : \neg b/d$ and $d_2 = \top : c/b$ and let $D = \{d_1, d_2\}$. The default theory $\langle W, D\rangle$ then has one extension: $\{a, b\}$. This extension can be generated by applying d_2 to W. The set $\{a, d, b\}$, which might seem to be possible to generate by applying d_1 and then d_2, is not an extension: b is derivable from this set, whereas b should not be derivable because the default rule d_1 with justification $\neg b$ was applied. The set $\{a, d\}$ is neither an extension, because it is not closed under the application of defaults. The rule d_2 is applicable, although application will yield a set that is not an extension. △

In the so-called *credulous* semantics for default logic a formula ϕ is said to follow from a default theory iff ϕ is in *one* of the extensions of this theory. The *sceptical* semantics defines that ϕ follows from a default theory iff ϕ is in *all* of the extensions of this theory.

4.2 Goal Base

In Section 4.2.1, we present a number of semantics of goal formulas, defined using the *goal base* of an agent configuration. Then, in Section 4.2.2, we investigate properties of these semantics, and we show how they are related to each other. In Section 4.3, we will consider a semantics for goals based on the goal base *and* the rule base, containing a set of goal adoption rules.

4.2.1 Semantics

In this section, we consider three semantics for goals. The first, which we call the basic semantics, assumes that the goal base of the agent is consistent. The

[3] That is, if W is consistent.

second is a proposal by Hindriks et al. [Hindriks et al., 2001], which allows the goal base to be inconsistent. The third semantics is a new proposal, which is a generalization of Hindriks' semantics.

The issue of *consistency* of goals is central with respect to semantics of goals. In agent theories as well as in agent programming frameworks, goals are often assumed or required to be consistent. The rationale is, that an agent should not simultaneously pursue situations that are mutually logically impossible.

In our framework, this can be modeled by requiring that the goal base of an agent configuration is consistent. Given this requirement, the semantics of goal formulas can be defined in a simple way as below, where $\mathbf{G}\phi$ is true in a configuration iff ϕ follows from the goal base in this configuration. The semantics of $\top$, negation and conjunction are defined analogously to the way this was done for belief formulas (Definition 4.3), but we omit this here and in definitions in the sequel for reasons of presentation.

Definition 4.6 *(basic ($\models_b$))* Let $\gamma \not\models \bot$.

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_b \mathbf{G}\phi \Leftrightarrow \gamma \models \phi$$

This semantics is similar to the semantics of goal formulas of Chapter 2 (Definition 2.7), except for two aspects. First, the semantics of Chapter 2 specifies that $\mathbf{G}\phi$ holds in a configuration if ϕ does not follow from the belief base in that configuration. This corresponds with a common perspective towards goals [Cohen and Levesque, 1990, van der Hoek et al., 1998, Hindriks et al., 2001] [van Riemsdijk et al., 2003b, Winikoff et al., 2002], which is that an agent should not have something as a goal that it already believes to be the case. In this chapter, we however omit this condition on beliefs from the semantics of goal formulas for reasons of simplicity. Second, we require in this definition that the goal base is consistent, which is in line with what is often required in agent theories and agent programming frameworks.

While we thus require the goal base to be consistent in the definition above, Hindriks argues in [Hindriks et al., 2001] that the goal base of the agent does not need to be consistent. Goals in the goal base do not have to be pursued simultaneously and could be achieved at different times. A goal base $\{p, \neg p\}$ should therefore be allowed in agent configurations. The semantics of goal formulas of Definitions 4.6 and 2.7 would in this case however have the undesirable characteristic that the inconsistent goal, i.e., the formula $\mathbf{G}\bot$, can be derived given an inconsistent goal base such as $\{p, \neg p\}$. Moreover, given that $\mathbf{G}\bot$ can be derived, any formula $\mathbf{G}\phi$ can be derived.

To avoid these undesired properties, Hindriks requires that individual goals in the goal base are consistent, rather than the goal base as a whole. He then defines the semantics of goal formulas as follows, specifying that $\mathbf{G}\phi$ holds in a configuration, iff there is a goal in the goal base of this configuration from which ϕ logically follows.

Definition 4.7 *(Hindriks ($\models_h$))* Let $\forall \phi \in \gamma : \phi \not\models \bot$.

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_h \mathbf{G}\phi \Leftrightarrow \exists \phi' \in \gamma : \phi' \models \phi$$

As an aside, we remark that this argument by Hindriks that goals in the goal base should be allowed to be inconsistent, without the agent being able to derive everything as a goal, can be viewed as related to research in the area of *paraconsistent* logics. Besnard and Hunter [Besnard and Hunter, 1995], e.g., observe that the need to reason with inconsistent information without the logic being *trivialized*, i.e., being able to derive anything on the basis of inconsistent information, is central to practical reasoning. In [Gabbay and Hunter, 1991], Gabbay and Hunter also argue that inconsistency should be viewed as a "good" thing, rather than as a "bad" thing. Allowing the representation of conflicting goals without trivializing the logic, will be an important concern throughout this chapter.

Returning to the semantics for goals as proposed by Hindriks, we observe that although this semantics allows for inconsistent goal bases without the possibility to derive the inconsistent goal, it can be considered too restrictive. Suppose for example that an agent has the (consistent) goal base $\{p, q\}$. In this case, one would most likely want the agent to derive that $p \wedge q$ is also a goal, i.e., that $\mathbf{G}(p \wedge q)$ holds. In particular, if the agent has an intention generation rule $\mathbf{G}(p \wedge q) \Rightarrow_{\mathbf{I}} \langle \pi, p \wedge q \rangle$, which represents the idea that the plan π is supposed to achieve a situation in which $p \wedge q$ holds, this rule should be applicable. If execution of the plan π is successful, both goals p and q of the agent would be achieved.

Moreover, if the agent has the inconsistent goal base $\{p, q, \neg p\}$, the given intention generation rule should *also* be applicable. If the plan π would achieve a situation in which $p \wedge q$ holds, part of the goals in the goal base would be achieved. The agent could then pursue the goal $\neg p$ consecutively.

Given these considerations, we propose the following semantic definition, which specifies that $\mathbf{G}\phi$ holds iff there is a consistent subset of the goal base from which ϕ follows.

Definition 4.8 *(consistent subset ($\models_s$))*

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_s \mathbf{G}\phi \Leftrightarrow \exists \gamma' \subseteq \gamma : (\gamma' \not\models \bot \text{ and } \gamma' \models \phi)$$

Note that, in contrast with Hindriks' semantics, we do not need to require that individual goals in the goal base are consistent. Inconsistent goals in the goal base are "ignored" by this definition, because we only consider subsets γ' of γ which are consistent ($\gamma' \not\models \bot$).[4] An inconsistent goal such as $p \wedge \neg p$, for example, cannot be used to derive $\mathbf{G}\bot$, or any other goal for that matter. Further, note

[4]The requirement could also be omitted from Definition 4.7 if the condition $\phi' \not\models \bot$ would be added to the righthand side of the definition, yielding a close resemblance with Definition 4.8. Definition 4.7 is however the one provided by Hindriks in [Hindriks et al., 2001].

that a formula $\mathbf{G}p \wedge \mathbf{G}\neg p$ is satisfiable under this semantics, without the formula $\mathbf{G}\bot$ being satisfiable (see also the explanation below Proposition 4.1).

This semantics can be viewed as related to a proposal by Poole [Poole, 1988] in the area of non-monotonic reasoning. He proposes to reason on the basis of a theory consisting of a set of facts, and a set of hypotheses (both being sets of first order formulas). A formula is then explainable on the basis of this theory, if it follows from the set of facts, and a consistent subset of the hypotheses.

Concluding this section, we make two remarks. First, we revisit the position that we addressed at the beginning of this section, which is that goals are often required to be consistent. For the proposed semantics of Definition 4.8, this is not required and the question now is, whether this can be justified.[5] We believe it can, for the following reason. As explained, the rationale behind the requirement of goal consistency is, that an agent should not simultaneously pursue goals that conflict. In our framework however, the intention base contains the plans of the agent, together with goals that are pursued by those plans. It is suggested by the semantics of intention generation of Definition 4.5, that intentions, i.e., the situations that are actively pursued by the agent, are *not* conflicting. An inconsistent goal base thus does not necessarily imply that inconsistent goals are simultaneously pursued. In a framework which allows all goals in the goal base to be pursued simultaneously, the consistency requirement for goals would indeed have to be adopted and the semantics of goal formulas of Definition 4.6 could then be used.

Secondly, we remark that this chapter considers representations of goals without any temporal information on the order in which the goals should be pursued. A representation of goals with a temporal component could be a way of reducing inconsistency, or, more accurately, of reducing what might *appear* to be an inconsistency without the temporal representation. Explorations along these lines are however not within the scope of this chapter.

4.2.2 Properties

In this section, we investigate properties of the semantics of Section 4.2.1, and compare these semantics to one another. In the proposition below, we use the notation $\models_{b,h,s}$ to indicate that a property holds under $\models_b$, $\models_h$, and $\models_s$.

[5] In the literature [Cohen and Levesque, 1990, Rao and Georgeff, 1991], *desires*, rather than goals, are often assumed or allowed to be inconsistent. One could thus argue that, since we allow goals to be inconsistent, we are actually formalizing desires, rather than goals. Motivational attitudes in cognitive agent programming languages are however often called goals, rather than desires. In order to stay in line with the common terminology, we will thus continue to use the term "goals", even if they are allowed to be inconsistent.

Proposition 4.1

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \quad \models_b \quad \mathbf{G}(\phi \rightarrow \psi) \rightarrow (\mathbf{G}\phi \rightarrow \mathbf{G}\psi) \tag{4.1}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \quad \models_{b,h,s} \quad \mathbf{G}(\phi \wedge \psi) \rightarrow (\mathbf{G}\phi \wedge \mathbf{G}\psi) \tag{4.2}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \quad \models_b \quad (\mathbf{G}\phi \wedge \mathbf{G}\psi) \rightarrow \mathbf{G}(\phi \wedge \psi) \tag{4.3}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \quad \models_{b,h,s} \quad \neg\mathbf{G}\bot \tag{4.4}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \quad \models_b \quad \neg(\mathbf{G}\phi \wedge \mathbf{G}\neg\phi) \tag{4.5}$$

Proof: Let $c = \langle \sigma, \gamma, \iota, \mathcal{R} \rangle$.
(4.1) We have to show that $c \models_b \mathbf{G}(\phi \rightarrow \psi) \rightarrow (\mathbf{G}\phi \rightarrow \mathbf{G}\psi)$. This means we have to show that $c \models_b \mathbf{G}(\phi \rightarrow \psi) \Rightarrow (c \models_b \mathbf{G}\phi \Rightarrow c \models_b \mathbf{G}\psi)$. Assume that $c \models_b \mathbf{G}(\phi \rightarrow \psi)$ and $c \models_b \mathbf{G}\phi$. This means that $\gamma \models \phi \rightarrow \psi$ and $\gamma \models \phi$. From this we can conclude that $\gamma \models \psi$, as which $c \models_b \mathbf{G}\psi$ is defined, yielding the desired result.
(4.2) We have to show that $c \models_b \mathbf{G}(\phi \wedge \psi) \Rightarrow c \models_b \mathbf{G}\phi \wedge \mathbf{G}\psi$. This means we have to show that $c \models_b \mathbf{G}(\phi \wedge \psi) \Rightarrow (c \models_b \mathbf{G}\phi$ and $c \models_b \mathbf{G}\psi)$, which is defined as $\gamma \models \phi \wedge \psi \Rightarrow (\gamma \models \phi$ and $\gamma \models \psi)$. The latter is obviously the case.

We have to show that $c \models_h \mathbf{G}(\phi \wedge \psi) \Rightarrow (c \models_h \mathbf{G}\phi$ and $c \models_h \mathbf{G}\psi)$. Assume that $c \models_h \mathbf{G}(\phi \wedge \psi)$, which is defined as: $\exists \phi' \in \gamma : \phi' \models \phi \wedge \psi$. From the latter we can conclude that $\exists \phi' \in \gamma : \phi' \models \phi$ and $\exists \phi' \in \gamma : \phi' \models \psi$, as which $c \models_h \mathbf{G}\phi$ and $c \models_h \mathbf{G}\psi$ is defined.

The proof for $\models_s$ is analogous.
(4.3) The proof is analogous to the proof of (4.2), for $\models_b$.
(4.4) We have to show that $c \models_b \neg\mathbf{G}\bot$, i.e., that $c \not\models_b \mathbf{G}\bot$, i.e., that $\gamma \not\models \bot$. This follows immediately, since γ is consistent.

We have to show that $c \not\models_h \mathbf{G}\bot$, i.e., that $\neg\exists \phi \in \gamma : \phi \models \bot$. Since each $\phi \in \gamma$ is consistent, this follows immediately.

We have to show that $c \not\models_s \mathbf{G}\bot$, i.e., that $\neg\exists \gamma' \subseteq \gamma : \gamma' \not\models \bot$ and $\gamma' \models \bot$. This is obviously the case.
(4.5) We have to show that $c \models_b \neg(\mathbf{G}\phi \wedge \mathbf{G}\neg\phi)$, i.e., that $c \not\models_b \mathbf{G}\phi \wedge \mathbf{G}\neg\phi$, i.e., that it is not the case that $c \models_b \mathbf{G}\phi$ and $c \models_b \mathbf{G}\neg\phi$, i.e., that it is not the case that $\gamma \models \phi$ and $\gamma \models \neg\phi$. This is the case, since γ is consistent. □

Property (4.1) corresponds with the **K** axiom of standard modal logics with possible worlds semantics (see, e.g., [Meyer and van der Hoek, 1995]). It expresses that goals are closed under classical logical consequence. The property is satisfied by the basic semantics, but not by Hindriks' semantics and the consistent subset semantics (contrary to what was claimed in [van Riemsdijk et al., 2005b], in which we stated that it is satisfied by the subset semantics). It is easy to see that Hindriks' semantics does not satisfy **K**. Take, e.g., a goal base $\{p, p \rightarrow q\}$. The formulas $\mathbf{G}p$ and $\mathbf{G}(p \rightarrow q)$ then hold, but $\mathbf{G}q$ does not hold. Under the consistent subset semantics, q would be a goal, since q follows from the

consistent set $\{p, p \rightarrow q\}$. An example explaining that the **K** axiom does not hold for the consistent subset semantics, is the following. Take a goal base $\{p \wedge r, (p \rightarrow q) \wedge \neg r\}$. In this case, $\mathbf{G}p$ and $\mathbf{G}(p \rightarrow q)$ hold, since they follow from the consistent sets $\{p \wedge r\}$ and $\{(p \rightarrow q) \wedge \neg r\}$, respectively. The formula $\mathbf{G}q$ however does not hold, since the set $\{p \wedge r, (p \rightarrow q) \wedge \neg r\}$ is not consistent, and can thus not be used to derive q.

Property (4.2) corresponds with the so-called **M** axiom, and can be used to axiomatize weaker modal logics (see, e.g., [Chellas, 1980]). It holds for all logics discussed in this section. Property (4.3) is the reverse of (4.2), and corresponds with the **C** axiom, which can also be used to characterize weaker modal logics [Chellas, 1980]. It expresses that separate goals can be combined into one. It does not hold in general for $\models_h$ and $\models_s$, since inconsistent formulas in the goal base may not be combined into one, in order to prevent the derivation of the inconsistent goal. The **M** and **C** axiom together are equivalent with the **K** axiom, i.e., instead of using **K** to axiomatize a modal logic, **M** and **C** can be used instead. It is thus perhaps not surprising to see that since **C** is not satisfied by $\models_h$ and $\models_s$, **K** is neither.

Property (4.4) corresponds with the **D** axiom of modal logics (see, e.g., [Meyer and van der Hoek, 1995]). It is satisfied by all semantics discussed in this section, and expresses that the inconsistent goal cannot be derived. This is an important property with respect to the representation of conflicting goals. Even though goals in the goal base may be inconsistent, we do not want to allow the derivation of the inconsistent goal, since this would trivialize the logic. The various semantics for goals as discussed in this chapter, which allow the representation of conflicting goals, have been designed to satisfy this property.

In standard modal logics, property (4.5) is equivalent with the **D** axiom, and is often also called **D**. In our logics, we can see that (4.4) and (4.5) cannot be used interchangeably, since (4.5) is not satisfied by Hindriks' semantics and the consistent subset semantics. A formula $\mathbf{G}p \wedge \mathbf{G}\neg p$, e.g., is satisfiable under these semantics (take, e.g., a goal base $\{p, \neg p\}$).

The next proposition expresses how the semantics of the previous section are related, if the goal base is consistent.

Proposition 4.2 Let $\gamma \not\models \bot$. Then the following holds.

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_b \mathbf{G}\phi \Leftrightarrow \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_s \mathbf{G}\phi \tag{4.6}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_h \mathbf{G}\phi \Rightarrow \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_b \mathbf{G}\phi \tag{4.7}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_h \mathbf{G}\phi \Rightarrow \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_s \mathbf{G}\phi \tag{4.8}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_b \neg\mathbf{G}\phi \Rightarrow \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_h \neg\mathbf{G}\phi \tag{4.9}$$

Proof: (4.6) If $\gamma \not\models \bot$, we have that $\exists \gamma' \subseteq \gamma : (\gamma' \not\models \bot$ and $\gamma' \models \phi)$ is equivalent with $\exists \gamma' \subseteq \gamma : \gamma' \models \phi$, which is equivalent with $\gamma \models \phi$. (4.7) If $\exists \phi' \in \gamma : \phi' \models \phi$, then $\gamma \models \phi$. (4.8) Follows immediately from (4.6) and (4.7). (4.9) If $\gamma \not\models \phi$, then $\neg\exists \phi' \in \gamma : \phi' \models \phi$. □

Property (4.6) states that under the assumption of consistency of the goal base, the basic semantics and the consistent subset semantics are equivalent. The set of goals, i.e., formulas ϕ for which $\mathbf{G}\phi$ holds in a configuration, is the same for the consistent subset semantics as for the basic semantics. If the goal base is consistent, we thus have that the **K** axiom and the various incarnations of the **D** axiom also hold for the consistent subset semantics. This is in line with work on BDI logics by Cohen and Levesque [Cohen and Levesque, 1990] and Rao and Georgeff [Rao and Georgeff, 1991], in which goals are assumed to be consistent, and also obey the **K** and **D** axioms.

Comparing the basic semantics with Hindriks' semantics, we see that the set of goals derivable under Hindriks' semantics is a subset of those derivable under the basic semantics (4.7). The opposite of (4.7) does not hold in general. Take, e.g., a goal base $\{p, q\}$. In that case, $\mathbf{G}(p \wedge q)$ holds under the basic semantics, but does not hold under Hindriks' semantics. We thus have that the basic semantics is properly stronger than Hindriks' semantics, if the goal base is consistent. Property (4.8) expresses that the set of goals derivable under Hindriks' semantics is a subset of those derivable under the consistent subset semantics. Since the consistent subset semantics is equivalent with the basic semantics, we can conclude that the consistent subset semantics is properly stronger than Hindriks' semantics.

The formulas ϕ such that $\neg\mathbf{G}\phi$ is true in a configuration, is the complement of the formulas for which $\mathbf{G}\phi$ holds. We thus have implication (4.9).

If we assume that each goal in the goal base is consistent, rather than demanding that the entire goal base is consistent, we also have that the consistent subset semantics is properly stronger than Hindriks' semantics. This can be concluded from the following proposition, and using the example where $\gamma = \{p, q\}$ to show that the proposition does not hold in the opposite direction.

Proposition 4.3 Let $\forall\phi \in \gamma : \phi \not\models \bot$. Then the following holds.

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_h \mathbf{G}\phi \Rightarrow \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_s \mathbf{G}\phi$$

Proof: If $\exists\phi' \in \gamma : \phi' \models \phi$, then $\exists\gamma' \subseteq \gamma : \gamma' \models \phi$ (let $\gamma' = \{\phi'\}$). □

Reflecting on Propositions 4.1 and 4.3, we can observe that the consistent subset semantics is properly stronger than Hindriks' semantics, but the properties considered in Proposition 4.1 do not distinguish the two. Finding a discriminating property is not a trivial task. It seems that a weaker version of (4.3) might be what we are looking for, since two goals may sometimes be combined into one, but not always. For example, given a goal base $\{p, q\}$, we have that $\mathbf{G}p$ and $\mathbf{G}q$ hold under $\models_s$, and $\mathbf{G}(p \wedge q)$ also holds (but $\mathbf{G}(p \wedge q)$ does not hold under $\models_h$). Given a goal base $\{p \wedge r, q \wedge \neg r\}$, we have that $\mathbf{G}p$ and $\mathbf{G}q$ hold, but $\mathbf{G}(p \wedge q)$ does *not* hold under $\models_s$, i.e., in this case, p and q may not be combined.

Since p and q may be combined in some but not all cases, it is not very likely that we can use a version of (4.3) in which we put conditions on ϕ and ψ

only. We might end up concluding that the strongest property we can come up with, is that (4.3) holds iff there is a consistent subset of γ from which both ϕ and ψ follow. This property is however not very informative, since it essentially repeats the semantic definition. Also, it is a property which is not very general, since it does not depend on general properties of γ or of ϕ and ψ. Further investigations along these lines are left for future research.

4.3 Goal Base and Goal Adoption Rules

In Section 4.2, we presented a number of semantics for goal formulas, based on the goal base of an agent configuration. In that setting, goals are thus not conditional. One cannot express, e.g., that the agent should adopt the goal to have a drink, if he is thirsty, i.e., one cannot express that goals sometimes depend on beliefs. Also, one cannot express that goals might depend on other goals. For example, one might want to express that if the agent believes he is at home, and has the goal to be in New York (assuming that he does not live in New York), should adopt the goal to be at the airport.

In order to provide for this kind of expressivity, we introduce so-called goal adoption rules in this section. In Section 4.3.1, we present a semantics for goal formulas, based on the goal base *and* these rules. As in the case of the goal base, goal adoption rules may also represent conflicting goals, and our semantics is designed to handle these conflicts without trivializing the logic. At the end of this section, we work out some simple examples. In Section 4.3.2, we investigate properties of our proposed semantics, and compare the semantics with the consistent subset semantics of Section 4.2.

4.3.1 Semantics

Below, we define the set of goal adoption rules. A goal adoption rule has a belief and goal condition as the antecedent and a propositional formula as the consequent. Intuitively, it means that if the belief and goal condition in the antecedent hold, the formula in the consequent can be adopted as a goal. As is argued in philosophical logic [Hansson, 1969], mental attitudes are conditional by nature.

Definition 4.9 *(goal adoption rule)* The set of goal adoption rules $\mathcal{R}_{\mathsf{GA}}$ is defined as follows:

$$\{\beta, \kappa \Rightarrow^{+}_{\mathbf{G}} \phi \mid \beta \in \mathcal{L}_{\mathsf{B}}, \kappa \in \mathcal{L}_{\mathsf{G}}, \phi \in \mathcal{L}\}.$$

These goal adoption rules were also proposed in [van Riemsdijk et al., 2005a]. In that paper, two kinds of semantics of these rules were given. One was based on the *application* of a goal adoption rule over a transition, comparable with the semantics of intention generation rules of Definition 4.5. The second kind,

and the one we focus on in this chapter, defined the semantics of goal *formulas*, given an agent configuration with a goal base and a set of goal adoption rules. This semantics was however defined for simple forms of goal adoption rules with only a belief condition. Further, it was very restrictive. Basically, a formula $\mathbf{G}\phi$ was specified to be true, if there was a goal adoption rule with true antecedent and a consequent equivalent to ϕ. The semantics for rules with a belief and goal condition was not specified. In this section, we propose a semantics for goal formulas that is based on goal adoption rules with belief *and* goal condition, which is less restrictive than the one provided in [van Riemsdijk et al., 2005a], and takes into account potential conflicts among goals.

The semantics we propose is based on default logic. The general idea is as follows. Goal adoption rules and the goals in the goal base are transformed into propositional default rules. This set of default rules has a number of (consistent) extensions, which represent the sets of compatible goals that an agent could derive on the basis of its rules and goal base. Given an agent configuration and an extension of the default rules generated from the goal adoption rules and goal base in this configuration, we define the semantics of goal formulas.

Default logic is designed to handle possibly conflicting defeasible rules. Goal adoption rules could be conflicting and default logic is therefore a natural way to interpret these rules. This can be illustrated by considering an agent with the following rules for deriving goals: if the agent believes that it's raining, it can derive the goal to take the bus, and if it has the goal to be on time, it can derive the goal not to take the bus (but to take a taxi instead, for example). Suppose the agent believes it is raining and wants to be on time, then it has a reason to derive the goal to take the bus *and* it has a reason to derive the goal not to take the bus. Rather than deriving both conflicting goals, default logic gives rise to two extensions or compatible goal sets: one containing the goal to take the bus and one not to take the bus.

Below, we define the function f that takes a set of goal adoption rules with only a goal condition and yields a set of propositional default rules. It will become clear later on why we define this function for rules without a belief condition. In the definition, we use CL to denote the set of conjunctions of goal literals. A goal literal is a formula of the form $\mathbf{G}\phi$ or $\neg\mathbf{G}\phi$, where $\phi \in \mathcal{L}$. Formulas of the former kind are called positive goal literals and formulas of the latter type negative goal literals. The formula $\top$ is treated as a positive goal literal. We use a function pl that takes a conjunction of goal literals and yields a set containing the propositional parts of the positive goal literals of this conjunction[6]. The function nl similarly yields the set of propositional parts of negative goal literals of a conjunction. Further, we use a function dnf that takes a set of goal adoption rules of the form $\kappa \Rightarrow^{+}_{\mathbf{G}} \phi$ and yields these rules with the antecedent transformed into disjunctive normal form. We map goal

[6]The propositional part of the positive goal literal $\top$ is the propositional formula $\top$. Also, if the number of positive goal literals is 0, the function pl yields the set $\{\top\}$.

adoption rules to disjunctive normal form, because rules of this form can be intuitively mapped to default rules. $\mathcal{R}_{\mathsf{GA}_{\mathsf{DNF}}}$ is the set of goal adoption rules with the antecedent in disjunctive normal form, i.e., $\mathcal{R}_{\mathsf{GA}_{\mathsf{DNF}}} = \{\bigvee_{1\leq i\leq n} cl_i \Rightarrow^{+}_{\mathbf{G}} \chi \mid n > 0, cl_i \in CL, \chi \in \mathcal{L}\}$. Finally, the number of elements in a set S is denoted by $|S|$.

Definition 4.10 *(goal adoption rules to default rules)* Let DR denote the set of propositional default rules. Let $cl, cl_1, \ldots, cl_k \in CL$. The function $t : \mathcal{R}_{\mathsf{GA}_{\mathsf{DNF}}} \rightarrow \wp(\mathsf{DR})$, taking a goal adoption rule and yielding a set of default rules, is then defined as follows, where $\phi_i \in pl(cl)$ for $1 \leq i \leq m$ and $\psi_j \in nl(cl)$ for $1 \leq j \leq n$ with $|pl(cl)| = m$ and $|nl(cl)| = n$, with $n \geq 0$. If $n = 0$, the sequence $\psi_1, \ldots, \psi_n$ is empty.

$$\begin{array}{lcl} t(cl \Rightarrow^{+}_{\mathbf{G}} \chi) & = & \{\phi_1 \wedge \ldots \wedge \phi_m : \neg\psi_1, \ldots, \neg\psi_n, \chi/\chi\} \\ t(cl_1 \vee \ldots \vee cl_k \Rightarrow^{+}_{\mathbf{G}} \chi) & = & \bigcup_{1\leq i\leq k} t(cl_i \Rightarrow^{+}_{\mathbf{G}} \chi) \end{array}$$

The function $f : \wp(\mathcal{R}_{\mathsf{GA}}) \rightarrow \wp(\mathsf{DR})$ taking a set of goal adoption rules of the form $\kappa \Rightarrow^{+}_{\mathbf{G}} \phi$ and yielding a set of default rules, is then defined as follows.

$$f(\mathsf{GA}) = \bigcup_{r \in dnf(\mathsf{GA})} t(r)$$

We explain this definition using an example. Consider the goal adoption rules $g_1 = \mathbf{G}p \wedge \neg\mathbf{G}q \Rightarrow^{+}_{\mathbf{G}} r$, $g_2 = \top \Rightarrow^{+}_{\mathbf{G}} p$ and $g_3 = \mathbf{G}r \Rightarrow^{+}_{\mathbf{G}} q$, corresponding with the default rules $d_1 = p : \neg q, r/r$, $d_2 = \top : p/p$ and $d_3 = r : q/q$, respectively.

When transforming a goal adoption rule with a conjunction as the antecedent, the propositional parts of positive goal literals are mapped onto the prerequisite of a default rule, whereas the propositional parts of negative goal literals are negated and mapped onto the justification of the default rule. This reflects the difference between for example the formulas $\mathbf{G}\neg q$ and $\neg\mathbf{G}q$: the former represents the *presence* of a goal $\neg q$, whereas the latter represents the *absence* of the goal q. Considering goal adoption rules g_1 and g_2, the set $\{p, r\}$ is an extension of the default rules d_1 and d_2. This reflects our intuition about goal adoption rules: p can be derived on the basis of the second rule and if p is a goal and q is not, we can derive goal r.

If we consider the default rules d_1, d_2 and d_3, we have that the set $\{p, r, q\}$ is *not* an extension of these rules. This is due to the fact that q, which was derived using rule d_3, is inconsistent with the justification $\neg q$ of rule d_1. This corresponds with our intuition about goal adoption rules: given rule g_1, r can only be a goal if q is not. The goals r and q thus cannot be part of the same extension.

Negative goal literals are mapped to a sequence of justifications, rather than to one conjunctive justification. The reason is, that we want to allow goal adoption rules such as $\neg\mathbf{G}p \wedge \neg\mathbf{G}\neg p \Rightarrow^{+}_{\mathbf{G}} q$, specifying that goal q can be adopted if neither p nor $\neg p$ is a goal. If we would map this rule to the default rule

$\top : p \wedge \neg p \wedge q/q$, we would get an inconsistent justification and the rule would never be applicable. The rule $\top : p, \neg p, q/q$ on the other hand does the job.

The consequent χ of a goal adoption rule is added to the justification, because we only want to derive a new goal if it is consistent with the already derived ones. Further, goal adoption rules without negative goal literals then yield so-called normal default rules, i.e., rules of the form $\phi : \chi/\chi$. Normal default rules have a number of desirable characteristics, such as the fact that normal default theories always have extensions [Brewka et al., 1997].

Moreover, a goal adoption rule such as $\mathbf{G}p \vee \mathbf{G}q \Rightarrow^{+}_{\mathbf{G}} r$ with a disjunctive goal formula in the antecedent is transformed into the set of *multiple* defaults $\{p : r/r, q : r/r\}$. The rationale is, that the goal r can be derived if either p or q is a goal. This is established through this set of default rules, because if p has been derived as a goal, the first rule can be applied to derive r. Alternatively, r can also be derived using the second rule if q has been derived as a goal.

The following function transforms goal bases to goal adoption rules and will be used in Definition 4.12. The reason for transforming the goal base into goal adoption rules in this way will be explained after Definition 4.12.

Definition 4.11 *(goal base to goal adoption rules)* The function $g : \wp(\mathcal{L}) \rightarrow \wp(\mathcal{R}_{\mathsf{GA}})$, taking a goal base and yielding a set of goal adoption rules, is defined as follows: $g(\gamma) = \{\top \Rightarrow^{+}_{\mathbf{G}} \phi \mid \phi \in \gamma\}$.

Note that the default rules corresponding with these goal adoption rules of the form $\top \Rightarrow^{+}_{\mathbf{G}} \phi$, have the form $\top : \phi/\phi$. Default rules of this form are often called Poole-type defaults, or supernormal defaults (see, e.g., [Poole, 1988, Brewka, 1991]).

In the definition of the semantics of goals below, we transform the goal adoption rules generated from the goal base, as well as the goal adoption rules in the rule base of the configuration, to default rules. That is, we only take those goal adoption rules for which the belief condition holds in the given configuration. These rules can be transformed into default rules by means of the function of Definition 4.10, if we remove the (true) belief condition. Given an extension of the generated default rules, we define that $\mathbf{G}\phi$ holds iff ϕ follows from this extension.

Definition 4.12 *(semantics of goals)* Let $\mathcal{R} = \langle \mathsf{IG}, \mathsf{GA} \rangle$, where $\mathsf{GA} \subseteq \mathcal{R}_{\mathsf{GA}}$ is a finite set of goal adoption rules. Let GA' be defined as

$$\{\kappa \Rightarrow^{+}_{\mathbf{G}} \phi \mid \exists (\beta, \kappa \Rightarrow^{+}_{\mathbf{G}} \phi) \in \mathsf{GA} : \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_{\mathcal{L}_{\mathsf{B}}} \beta\}.$$

Let E be an extension of $\langle \emptyset, f(g(\gamma)) \cup f(\mathsf{GA}') \rangle$. The default semantics $\models^{E}_{d}$ for goal formulas in the presence of these goal adoption rules, given the extension E, is then as follows.

$$\begin{array}{lcl}
\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models^{E}_{d} \mathbf{G}\phi & \Leftrightarrow & E \models \phi \\
\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models^{E}_{d} \neg\kappa & \Leftrightarrow & \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \not\models^{E}_{d} \kappa \\
\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models^{E}_{d} \kappa \wedge \kappa' & \Leftrightarrow & \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models^{E}_{d} \kappa \text{ and } \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models^{E}_{d} \kappa'
\end{array}$$

Note that the set of facts, i.e., the first component of a default theory, is empty in our case. In the sequel, we will therefore omit the set of facts and speak of extensions of a set of default rules. We chose to transform the goal base into default rules, rather than taking these as facts and considering extensions of $\langle \gamma, f(\mathsf{GA}') \rangle$. The reason is, that we want to allow γ to be inconsistent. A default theory with an inconsistent set of facts only has one extension, i.e., the inconsistent extension. This is undesirable, since we want to allow the representation of inconsistent goals, without trivializing the logic (see Section 4.2.1).

An alternative way of defining the default semantics could be the following: $c \models_d \mathbf{G}\phi \Leftrightarrow \exists E : E \models \phi$, where E is an extension of configuration c. Rather than parameterizing the semantics with an extension of c as in Definition 4.12, this definition incorporates an existential quantification over the extensions in the definition of the semantics. This definition is analogous to the consistent subset semantics, in the sense that both definitions use existential quantification over a consistent set of formulas. We, however, chose to define the default semantics as in Definition 4.12, in order to obtain the validity of Theorem 4.1, which will be presented in Section 4.3.2.

We conclude this section by presenting two simple examples, in order to give some idea about the kinds of situations which can be modeled using goal adoption rules. The first example is about an agent which has to carry cargo from a source location to a target location.

Example 4.2 *(carrying cargo)* Consider that one wants to express that if the agent is at the location of the source, it should have the goal to have cargo, and if it is at the target, it should have the goal not to have cargo. Further, if the agent believes he is at the source and he has cargo, he should have the goal to be at the target. Finally, if he believes he is at the source, and has the goal to be at the target, he should have the goal to be at some waypoint[7] in-between the source and the target. This can be useful if the agent only has plans to get from the source to the waypoint, and from the waypoint to the target (see Example 4.6). This could be modeled using the following goal adoption rules.

$$
\begin{array}{lll}
\mathbf{B}(\mathit{source} \wedge \neg \mathit{haveCargo}) & \Rightarrow^{+}_{\mathbf{G}} & \mathit{haveCargo} \\
\mathbf{B}(\mathit{target} \wedge \mathit{haveCargo}) & \Rightarrow^{+}_{\mathbf{G}} & \neg \mathit{haveCargo} \\
\mathbf{B}(\mathit{source} \wedge \mathit{haveCargo}) & \Rightarrow^{+}_{\mathbf{G}} & \mathit{target} \\
\mathbf{B}(\mathit{source}), \mathbf{G}(\mathit{target}) & \Rightarrow^{+}_{\mathbf{G}} & \mathit{waypoint}
\end{array}
$$

△

This example illustrates that goals might be conditional on beliefs, and also on other goals. The fourth rule is an example of the specification of the adoption

[7] According to *The American Heritage: Dictionary of the English Language*, a waypoint is a point between major points on a route, as along a track.

of landmarks, as it was called in [van Riemsdijk et al., 2005a]. A landmark is a goal which the agent has to achieve, on its way to achieving another goal. In this example, the waypoint can be viewed as a landmark, which the agent has to achieve in order to achieve the goal of being at the target.

The next example is about an agent wanting either tea or coffee.

Example 4.3 *(tea or coffee)* Consider that one wants to express that if an agent is thirsty, he should either have the goal to have coffee or to have tea, but not both, i.e., if the agent does not already have the goal to have tea, it can adopt the goal to have coffee, and vice versa. This could be modelled using the following goal adoption rules.

$$\begin{array}{lll} \mathbf{B}(\mathit{thirsty}), \neg\mathbf{G}(\mathit{tea}) & \Rightarrow^{+}_{\mathbf{G}} & \mathit{coffee} \\ \mathbf{B}(\mathit{thirsty}), \neg\mathbf{G}(\mathit{coffee}) & \Rightarrow^{+}_{\mathbf{G}} & \mathit{tea} \end{array}$$

Assuming the agent indeed believes he is thirsty, the default rules corresponding with these goal adoption rules are $\top : \neg \mathit{tea}, \mathit{coffee} / \mathit{coffee}$ and $\top : \neg \mathit{coffee}, \mathit{tea} / \mathit{tea}$. There are two extensions of these default rules, i.e., $\{\mathit{coffee}\}$ and $\{\mathit{tea}\}$. △

This example illustrates the use of negative goal literals in goal adoption rules, i.e., they can be used to express that certain goals are *incompatible*, which means they should not be part of the same extension. Using negative goal literals thus provides a way to express that certain goals are incompatible, even though they are logically consistent. The idea is that incompatible goals should not be pursued simultaneously. This is discussed further in Section 4.4.

4.3.2 Properties

In this section, we investigate some properties of the default semantics of goals. In the sequel, we will use the following definition of extension of a configuration.

Definition 4.13 *(extension of a configuration)* Let $\mathcal{R} = \langle \mathsf{IG}, \mathsf{GA} \rangle$ and let $c = \langle \sigma, \gamma, \iota, \mathcal{R} \rangle$ be an agent configuration. Let GA' be $\{\kappa \Rightarrow^{+}_{\mathbf{G}} \phi \mid \exists (\beta, \kappa \Rightarrow^{+}_{\mathbf{G}} \phi) \in \mathsf{GA} : c \models_{\mathcal{L}_{\mathsf{B}}} \beta\}$. E is then said to be an extension of the configuration c, iff E is an extension of $f(g(\gamma)) \cup f(\mathsf{GA}')$.

The first theorem specifies the following: if a configuration contains a goal adoption rule of which the antecedent is true given an extension of the defaults generated on the basis of this configuration, and the consequent of this rule is consistent with this extension, then the consequent is a goal in this configuration. This theorem formalizes an important desired characteristic of the semantics of goals, being that if the antecedent of a goal adoption rule holds, the consequent is a goal.

Theorem 4.1 Let $c = \langle \sigma, \gamma, \iota, \mathcal{R} \rangle$ be a configuration, let $\mathcal{R} = \langle \mathsf{IG}, \mathsf{GA} \rangle$ and let E be an extension of c. Then the following holds.

$$\text{If } \exists (\beta, \kappa \Rightarrow^{+}_{\mathbf{G}} \phi) \in \mathsf{GA} : \big(c \models_{\mathcal{L}_{\mathsf{B}}} \beta \text{ and } c \models^{E}_{d} \kappa \text{ and } E \not\models \neg\phi\big) \text{ then } c \models^{E}_{d} \mathbf{G}\phi.$$

Proof: Let $\kappa = \bigvee_{1 \leq k \leq o} cl_k$ with $o > 0, cl_k \in CL$. Since $c \models_d^E \kappa$ by assumption, it must be the case by Definition 4.12 that $c \models_d^E cl_k$, for some $1 \leq k \leq o$. Assume that $c \models_d^E cl_k$ and let $\phi_i \in pl(cl_k)$ for $1 \leq i \leq m$ and $\psi_j \in nl(cl_k)$ for $1 \leq j \leq n$ with $|pl(cl_k)| = m$ and $|nl(cl_k)| = n$. If $c \models_d^E cl_k$, it must be the case by Definition 4.12, that $E \models \bigwedge_{1 \leq i \leq m} \phi_i$ and $E \not\models \psi_j$ for $1 \leq j \leq n$. We also have that $E \not\models \neg\phi$. The default $\phi_1 \wedge \ldots \wedge \phi_m : \neg\psi_1, \ldots, \neg\psi_n, \phi/\phi$ is thus applicable to E. As E is closed under the application of applicable defaults, it must be the case that $E \models \phi$ and thus by Definition 4.12, we can conclude that $c \models_d^E \mathbf{G}\phi$. □

As stated below Definition 4.12, this theorem would not hold if we would have defined the default semantics analogously to the consistent subset semantics, i.e., as $c \models_d \mathbf{G}\phi \Leftrightarrow \exists E : E \models \phi$. Consider, for example, a set of goal adoption rules $\{\neg\mathbf{G}p \Rightarrow_{\mathbf{G}}^{+} q,\ \neg\mathbf{G}q \Rightarrow_{\mathbf{G}}^{+} p,\ \mathbf{G}p \wedge \mathbf{G}q \Rightarrow_{\mathbf{G}}^{+} r\}$ with the set of corresponding default rules $\{\top : \neg p, q/q,\ \top : \neg q, p/p,\ p \wedge q : r/r\}$. This set of default rules has two extensions, i.e., $\{q\}$ and $\{p\}$. The first extension results from the application of the first default rule, after which the second cannot be applied, and vice versa for the second extension. The third default rule cannot be applied, since $p \wedge q$ does not follow from either of these extensions.

Under the suggested alternative semantics, we would thus have that $\mathbf{G}p \wedge \mathbf{G}q$ holds, but that $\mathbf{G}r$ does not hold. This can be considered counterintuitive, considering the third goal adoption rule. If we evaluate goal formulas always with respect to a single extension, as done in the semantics of Definition 4.12, the formula $\mathbf{G}p \wedge \mathbf{G}q$ would not hold, since p and q are goals in different extensions. Our proposed semantics thus prevents this kind of unintuitive behavior from occurring.

The following proposition expresses whether the properties as established in Proposition 4.1 for (some of) the semantics of Section 4.2.1, also hold for the default semantics.

Proposition 4.4

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_d^E \mathbf{G}(\phi \rightarrow \psi) \rightarrow (\mathbf{G}\phi \rightarrow \mathbf{G}\psi) \tag{4.10}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_d^E (\mathbf{G}\phi \wedge \mathbf{G}\psi) \leftrightarrow \mathbf{G}(\phi \wedge \psi) \tag{4.11}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_d^E \neg\mathbf{G}\bot \tag{4.12}$$

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_d^E \neg(\mathbf{G}\phi \wedge \mathbf{G}\neg\phi) \tag{4.13}$$

Proof: Let $c = \langle \sigma, \gamma, \iota, \mathcal{R} \rangle$.
(4.10) We have to show that $c \models_d^E \mathbf{G}(\phi \rightarrow \psi) \rightarrow (\mathbf{G}\phi \rightarrow \mathbf{G}\psi)$. This means we have to show that $c \models_d^E \mathbf{G}(\phi \rightarrow \psi) \Rightarrow (c \models_d^E \mathbf{G}\phi \Rightarrow c \models_d^E \mathbf{G}\psi)$. Assume that $c \models_d^E \mathbf{G}(\phi \rightarrow \psi)$ and $c \models_d^E \mathbf{G}\phi$. This means that $E \models \phi \rightarrow \psi$ and $E \models \phi$ (Definition 4.12). From this we can conclude that $E \models \psi$, which is equivalent with $c \models_d^E \mathbf{G}\psi$, yielding the desired result.

(4.11) We have to show that $c \models_d^E \mathbf{G}\phi \wedge \mathbf{G}\psi \Leftrightarrow c \models_d^E \mathbf{G}(\phi \wedge \psi)$. This means we have to show that ($c \models_d^E \mathbf{G}\phi$ and $c \models_d^E \mathbf{G}\psi$) $\Leftrightarrow c \models_d^E \mathbf{G}(\phi \wedge \psi)$, which is defined as ($E \models \phi$ and $E \models \psi$) $\Leftrightarrow E \models \phi \wedge \psi$ (Definition 4.12). This is obviously the case.
(4.12) We have to show that $c \models_d^E \neg\mathbf{G}\bot$, i.e., that $c \not\models_d^E \mathbf{G}\bot$, i.e., that $E \not\models \bot$. This follows immediately from the fact that E is consistent.
(4.13) We have to show that $c \models_d^E \neg(\mathbf{G}\phi \wedge \mathbf{G}\neg\phi)$, i.e., that $c \not\models_d^E \mathbf{G}\phi \wedge \mathbf{G}\neg\phi$. This means we have to show that it is not the case that $c \models_d^E \mathbf{G}\phi$ and $c \models_d^E \mathbf{G}\neg\phi$, which is defined as $E \models \phi$ and $E \models \neg\phi$. We have that E is consistent (Section 4.1.2), which means that this is a contradiction, yielding the desired result. □

As we can see, all properties of Proposition 4.1, and thus those holding for the basic semantics, also hold for the default semantics. This is perhaps not surprising, since goal formulas evaluated under the default semantics are evaluated with respect to a single extension, and extensions are consistent. Formulas evaluated under the basic semantics are also evaluated with respect to a consistent set of formulas, i.e., the goal base.

The fact that goal formulas evaluated under the default semantics and under the basic semantics obey the same logical properties, does *not* imply that these semantics are equivalent: the extension considered under the default semantics does not have to be equal to the goal base. It is thus not the case that if ϕ is a goal under the default semantics, that ϕ is also a goal under the basic semantics. Nevertheless, if the goal base is consistent, and the set of goal adoption rules is empty, we *do* have that the basic semantics and the default semantics are equivalent, as expressed by the proposition below. This is intuitive, since if the goal base is consistent, there is precisely one extension of the default rules corresponding with the goal base. This extension is equivalent with the goal base, i.e., the same sets of propositional formulas are derivable from these sets. By Proposition 4.2, we have that the default semantics is also equivalent with the consistent subset semantics in this case, and is stronger than Hindriks' semantics.

Proposition 4.5 Let $\gamma \not\models \bot$, let $\mathcal{R} = \langle \mathsf{IG}, \emptyset \rangle$, and let E be an extension of $\langle \sigma, \gamma, \iota, \mathcal{R} \rangle$. We then have the following.

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_b \mathbf{G}\phi \Leftrightarrow \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_d^E \mathbf{G}\phi$$

Proof: Let $c = \langle \sigma, \gamma, \iota, \mathcal{R} \rangle$. We have that $c \models_b \mathbf{G}\phi$ is defined as $\gamma \models \phi$. If γ is consistent, there is only one extension of $\langle \emptyset, f(g(\gamma)) \rangle$, and it is equivalent with γ. Since $c \models_d^E \mathbf{G}\phi$ is defined as $E \models \phi$, we have that $c \models_d^E \mathbf{G}\phi \Leftrightarrow \gamma \models \phi$, which is equivalent with the definition of $c \models_b \mathbf{G}\phi$. □

If we drop the requirement that the goal base is consistent, but keep the requirement that the set of goal adoption rules is empty, we can relate the consistent

subset semantics and the default semantics as expressed in the following theorem.

Theorem 4.2 Let $\mathcal{R} = \langle \mathsf{IG}, \emptyset \rangle$. Then the following holds.

$$\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_s \mathbf{G}\phi \Leftrightarrow \exists E : \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_d^E \mathbf{G}\phi$$

In the proof of this theorem we use the following lemma, in which the notion of a maximal consistent subset is used. A set of propositional formulas γ' is a maximal consistent subset of a set of formulas γ iff $\gamma' \subseteq \gamma$, $\gamma' \not\models \bot$ and $\neg\exists \phi \in \gamma : \phi \notin \gamma'$ and $\{\phi\} \cup \gamma' \not\models \bot$.

Lemma 4.1 There is a consistent subset γ' of γ such that $\gamma' \models \phi$ iff there is a maximal consistent subset γ' of γ such that $\gamma' \models \phi$. Further, γ' is a maximal consistent subset of γ iff γ' is an extension of $\{\top : \phi/\phi \mid \phi \in \gamma\}$ [Brewka et al., 1997].

Proof of Theorem 4.2: By Definition 4.12 and the fact that $\mathsf{GA} = \emptyset$, E must be an extension of $f(g(\gamma))$. $\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_s \mathbf{G}\phi$ means that there is a consistent subset γ' of γ such that $\gamma' \models \phi$. By Lemma 4.1, this is equivalent to there being a maximal consistent subset γ' of γ such that $\gamma' \models \phi$. We thus have to show that there is a maximal consistent subset γ' of γ such that $\gamma' \models \phi$ iff there is an extension E of $f(g(\gamma))$ such that $E \models \phi$.[8] By Definition 4.11, we have that $g(\gamma) = \{\top \Rightarrow^+_{\mathbf{G}} \phi \mid \phi \in \gamma\}$ and therefore $f(g(\gamma)) = \{\top : \phi/\phi \mid \phi \in \gamma\}$. By Lemma 4.1, we then have that γ' is a maximal consistent subset of γ iff γ' is an extension of $f(g(\gamma))$, yielding the desired result. □

The default semantics and consistent subset semantics are thus "equivalent" in some sense. The difference between the two is that the existential quantifier is incorporated in the semantics in case of the consistent subset semantics. As explained above, the existential quantifier was not incorporated in the default semantics, to yield the validity of Theorem 4.1. In order to relate the two semantics, we thus have to add the existential quantification, as done in Theorem 4.2. If we would have defined the default semantics with the existential quantifier incorporated, the default semantics and consistent subset semantics would have been truly equivalent. That is, if the set of goal adoption rules is empty.

Theorem 4.2 does not hold for arbitrarily composed goal formulas, for similar reasons. A formula $\mathbf{G}\phi \wedge \mathbf{G}\neg\phi$ is satisfiable under $\models_s$, but not under $\models_d^E$ (Propositions 4.1 and 4.4). Goal formulas evaluated under $\models_d^E$ are evaluated under one extension, whereas each conjunct of a conjunction evaluated under $\models_s$, can be evaluated with respect to a *different* consistent subset. It can thus be the case that a formula such as $\mathbf{G}\phi \wedge \mathbf{G}\neg\phi$ holds under $\models_s$, but does not hold under $\models_d^E$ for some E.

[8] A similar proposition was used, although not proven, by Reiter in [Reiter, 1987].

The last proposition of this section specifies a property of the translation of goal adoption rules into default rules, with respect to conjunctions of positive goal literals in the antecedent.

Proposition 4.6 A goal adoption rule r of the form $\bigwedge_{1\leq i\leq m} \mathbf{G}\phi_i \Rightarrow^{+}_{\mathbf{G}} \chi$ is equivalent with the rule r' of the form $\mathbf{G}(\bigwedge_{1\leq i\leq m} \phi_i) \Rightarrow^{+}_{\mathbf{G}} \chi$, i.e., $f(\{r\}) = f(\{r'\})$.

Proof: Immediate from Definition 4.10. □

4.4 Dynamics of Goals and Intentions

So far, we have discussed goal adoption and intention generation in a *static* way. That is, we have addressed how, given some configuration, we can define the goals of an agent (Sections 4.2 and 4.3), and what intentions might be generated in this configuration on the basis of these goals (Section 4.1.1). However, when considering the *dynamic* behavior of an agent over time, i.e., the agent's configuration changing from one to another continuously, a number of issues arise which we have not addressed so far.

We divide these issues into two (not completely disjoint) categories, i.e., issues related to commitment strategies for goals (Section 4.4.1), and issues related to intention generation (Section 4.4.2). It turns out that both kinds of issues become more intricate when it comes to agents with goal adoption rules, compared with agents with only a goal base.

4.4.1 Commitment Strategies for Goals

A commitment strategy for goals expresses when an agent may *drop* a goal it has formed previously [Rao and Georgeff, 1991, Winikoff et al., 2002] [van Riemsdijk et al., 2005a]. One type of commitment strategy is called *blind* commitment [Rao and Georgeff, 1991][9]. A blindly committed agent maintains its goals until it believes it has achieved them. Goals like this which should be dropped once they are achieved, are generally called *achievement goals*. One can also distinguish so-called *maintenance goals*, which express a situation that the agent should maintain, i.e., the agent should make sure that the situation expressed by the maintenance goal holds continuously (see, e.g., [Pokahr et al., 2005b] for an implementation).

In [Hindriks et al., 2001], and following that paper in [van Riemsdijk et al., 2003b, Dastani et al., 2004], blindly committed agents are

[9]The cited paper on BDI logic actually proposes commitment strategies for intentions, rather than for goals. In agent programming, commitment strategies have however also been used in the context of goals (see, e.g., [Winikoff et al., 2002, van Riemsdijk et al., 2005a]).

implemented as follows. The goals are implemented using a goal base, as in Section 4.2. On the basis of these goals, plans or actions are selected and executed. After the execution of an action, it is checked whether goals in the goal base are (believed to be) reached. If so, they are removed from the goal base. For example, consider a goal base $\{p, p \wedge q\}$. If p is believed to be reached (and q is not), p is removed from the goal base. The goal $p \wedge q$ is not removed (nor updated), since this goal expresses that the agent should achieve both p and q at the same time.

This mechanism is a simple and intuitive implementation of blind commitment: a goal is dropped, only if believed to be achieved. When it comes to agents with goal adoption rules however, the implementation of blind commitment is less straightforward. In principle, one could remove a goal adoption rule of the form $\beta, \kappa \Rightarrow^{+}_{\mathbf{G}} \phi$, if ϕ is believed to be achieved. Intuitively, one could however argue that rules express some kind of general "knowledge" of the agent about which goals it should adopt, and that rules should thus *not* be removed.

On the other hand, if rules are maintained, one could argue that these rules actually model maintenance goals, rather than achievement goals. However, since goal adoption rules have a condition on beliefs and goals, maintaining rules does not necessarily mean that ϕ is always a goal of the agent. That is, it can be the case that ϕ is a goal in a certain configuration and not in the next, because, e.g., the beliefs of the agent have changed. Consequently, maintaining rules does not mean that we are modeling maintenance goals.

The question now is, whether we are modeling achievement goals, i.e., goals that are dropped once they are achieved, through maintaining the goal adoption rules. This is neither the case, since ϕ might be a goal in one configuration and not in the next (in which case we could say that ϕ has been dropped [van Riemsdijk et al., 2005a]), because the belief condition of the goal adoption rule with ϕ in its consequent has become false. Another way to put this, is that the modeling of goals using goal adoption rules results in the level of persistency of goals possibly being low, or at least lower than for blindly committed agents (see also [Winikoff et al., 2002]). This can be considered undesirable. However, we suggest to embed our modeling of goals in a framework in which the agent also has an intention base. The semantics of intentions could then be defined such that the persistency of intentions is higher, i.e., one could implement an agent that is blindly committed towards his intentions.[10]

A final issue we mention with respect to commitment strategies for goals, is concerned with the case in which goal adoption rules are removed once the goal in the consequent is achieved. In order to explain this, we revisit Example 4.4. This example aims to express a situation in which the agent should get either tea or coffee. If the agent chooses to get coffee, the goal adoption rule for coffee can be removed (assuming the plan for getting coffee has succeeded). The agent could then however go for tea the next point in time. This does not do justice

[10] This would actually be more in line with the proposal in [Rao and Georgeff, 1991].

to the idea that the agent should get either tea or coffee. This problem could be solved by modeling the situation using a more complex goal, as suggested below Example 4.3. That approach however also has its disadvantages, as will be discussed in Section 4.4.2. On the other hand, one could argue that rules should not be removed in this example in the first place, since the rules aim to express that if the agent is thirsty, he should *always* get either tea or coffee. That is, the belief condition guards the adoption of the goals, and rules may thus be maintained.

This discussion illustrates that it is not immediately obvious how to implement appropriate commitment strategies for an agent with goal adoption rules. It will probably depend on the context, which is the most appropriate implementation. Nevertheless, we conjecture that in most cases it will be most intuitive to maintain the goal adoption rules, and to implement an appropriate commitment strategy for the intentions.

4.4.2 Intention Generation

Once the goals of an agent have been defined, the agent should act towards achieving these goals. That is, the agent should form plans, by means of which its goals can be achieved. In this chapter, we do consider the generation of plans only, but we attach the goal for which a plan was generated to the plan. This is done in order to be able to implement a screen of admissibility (Section 4.1.1). The resulting plan-goal pair is what we refer to as intentions.[11] There are various ways in which the agent can generate intentions on the basis of its goals. One possibility is to use a planning algorithm. Considering our proposal of Section 4.3, the agent could, e.g., choose one extension and use this as input to a planner.

Another approach, which is more in line with current work on agent programming languages, is to let the programmer specify which intentions the agent can adopt for which goals. This can be done using intention generation rules, as were introduced in Section 4.1.1. An advantage of this approach is, compared with planning, the lower computational complexity. It does however put a heavier burden on the programmer, who has to design effective intention generation rules.

Given our intuitions for a construct of intention generation rules, i.e., rules which specify which intention may be selected for which goal, it is important to investigate the precise semantics of these rules. A first suggestion that abstracts from a particular semantics of goals, was already done in Section 4.1.1 (Definition 4.5). That definition can be used in the context of the semantics of

[11] In agent programming, plans are often referred to as intentions (see, e.g., [Rao, 1996]), while in logics in which intentions are formalized, intentions are usually declarative [Rao and Georgeff, 1991]. We combine both aspects, and refer to the combination as "intention".

goals of Section 4.2, by simply plugging one of these semantics of goals into the definition for intention generation.

As will be discussed below, the same strategy however does not yield satisfactory results for intention generation in the case of the default semantics for goals. In the sequel, we propose an alternative that solves an important problem with respect to this semantics for intention generation. Nevertheless, there are a number of issues that the alternative semantics still does not address. These will also be discussed briefly below, but investigating ways of taking these into account in the semantics is left for future research.

Semantics of Intention Generation

A first attempt at defining the semantics of intention generation, that straightforwardly adapts the semantics of Definition 4.5 and that was also suggested in [van Riemsdijk et al., 2005b], is the following.

Definition 4.14 *(semantics of intention generation)* Let $c = \langle \sigma, \gamma, \iota, \mathcal{R} \rangle$ be a configuration, let $\mathcal{R} = \langle \mathsf{IG}, \mathsf{GA} \rangle$ and let $\beta, \kappa \Rightarrow_{\mathbf{I}} \langle \pi, \phi \rangle \in \mathsf{IG}$ be an intention generation rule. The semantics of applying this rule is then as follows, where $\iota' = \iota \cup \{\langle \pi, \phi \rangle\}$ and $\delta = \{\phi \mid \langle \pi, \phi \rangle \in \iota\}$.

$$\frac{c \models_{\mathcal{L}_{\mathsf{B}}} \beta \qquad \exists E : c \models_d^E \kappa \qquad \{\phi\} \cup \delta \not\models \bot}{\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \rightarrow \langle \sigma, \gamma, \iota', \mathcal{R} \rangle}$$

This semantics of intention generation specifies that an intention generation rule is applicable if an extension of the relevant configuration exists under which the goal condition holds. The definition is a variant of Definition 4.5, where the goal condition of an intention generation rule is evaluated under the default semantics.

In this definition, we choose to specify that an intention generation rule can be applied if an extension *exists* under which the goal condition holds, rather than defining that a rule can be applied if the goal condition holds under *all* extensions of the configuration. Our proposal thus corresponds with the credulous semantics of default logic. This is based on the intuition that the agent can choose to try to reach any of the goals it has, even if it conflicts with another goal of his. It can just not fulfill these conflicting goals at the same time. In principle, one could also define a skeptical version of intention generation in a similar way, if this is desirable in a certain context.

An advantage of this definition of intention generation, which comes with our definition of the semantics of goals, is that intentions are generated on the basis of a compatible set of goals. The goal condition of an intention generation rule is evaluated with respect to a *single* extension. An intention generation rule $\mathbf{G}p \wedge \mathbf{G}q \Rightarrow_{\mathbf{I}} \langle \pi, \phi \rangle$, for example, can thus not be applied if p is a goal in one extension, and q is a goal in another (and not in the first). This is desirable, since the fact that the specification of goal adoption rules has resulted in p and

q being in different extensions, represents that these goals are incompatible, and should not be pursued simultaneously (see Example 4.3). In particular, an intention generation rule such as $\mathbf{G}p \wedge \mathbf{G}\neg p \Rightarrow_{\mathbf{I}} \langle \pi, \phi \rangle$ will never be applicable, since p and $\neg p$ cannot be part of the same extension.

Continuing this line of reasoning, one could argue that it should not only be the case that a single intention is selected on the basis of compatible goals, but also that the intentions in the intention base should be mutually compatible. This could be realized by defining that a newly generated intention should be compatible with existing ones, thereby having the existing intentions behave as a screen of admissibility, as discussed in Section 4.1.1. This screen of admissibility function is provided in part by the third condition in the antecedent of the transition rule of Definition 4.14, since mutually inconsistent intentions are clearly incompatible. This condition, however, does not always prevent goals from different extensions, which are thus incompatible, from being pursued simultaneously. We illustrate this using the following example.

Example 4.4 *(coffee and tea (continued))* Consider the situation of Example 4.3, in which the agent should get either coffee or tea, if it is thirsty. We repeat the goal adoption rules of that example here. Further, assume the agent has two intention generation rules, i.e., one for getting coffee and one for getting tea, where *getCoffee* and *getTea* are plans for getting coffee and tea, respectively.

$$\begin{array}{lll} \mathbf{B}(\mathit{thirsty}), \neg\mathbf{G}(\mathit{tea}) & \Rightarrow^{+}_{\mathbf{G}} & \mathit{coffee} \\ \mathbf{B}(\mathit{thirsty}), \neg\mathbf{G}(\mathit{coffee}) & \Rightarrow^{+}_{\mathbf{G}} & \mathit{tea} \\ \mathbf{G}(\mathit{coffee}) & \Rightarrow_{\mathbf{I}} & \langle \mathit{getCoffee}, \mathit{coffee} \rangle \\ \mathbf{G}(\mathit{tea}) & \Rightarrow_{\mathbf{I}} & \langle \mathit{getTea}, \mathit{tea} \rangle \end{array}$$

Assume the agent is thirsty, and assume it decides to apply the first intention generation rule, yielding intention base $\{\langle \mathit{getCoffee}, \mathit{coffee} \rangle\}$. In this situation, the agent is still thirsty, and both the goal to have coffee and the goal to have tea can thus be derived. This means in particular, that the second intention generation rule is applicable in this situation, and an application of this rule will yield the intention base $\{\langle \mathit{getCoffee}, \mathit{coffee} \rangle, \langle \mathit{getTea}, \mathit{tea} \rangle\}$. The semantics of intention generation rules thus undermines the implementation of the idea of incompatible goals, as modeled using goal adoption rules. △

Note that the requirement of consistency as used in Definitions 4.5 and 4.14 is sufficient when considering the semantics of Section 4.2. The only way in which goals or intentions may be incompatible in those semantics, is by being inconsistent. The third condition of the transition rule for intention generation properly prevents the adoption of intentions which are inconsistent with existing ones.

Before we come to our solution of the problem as discussed in Example 4.4, we make a side remark about the modeling of the "tea or coffee" scenario. In principle, one could also model it using the following goal adoption rule:

$\mathbf{B}(\mathit{thirsty}) \Rightarrow^{+}_{\mathbf{G}} (\mathit{coffee} \vee \mathit{tea}) \wedge \neg(\mathit{coffee} \wedge \mathit{tea})$, giving rise to the single extension $\{(\mathit{coffee} \vee \mathit{tea}) \wedge \neg(\mathit{coffee} \wedge \mathit{tea})\}$. One could however argue that it is more involved to design intention generation rules for complex goals like this. One would then need to define a plan which results in the agent either getting coffee or tea,[12] and which could be selected if it has the goal $(\mathit{coffee} \vee \mathit{tea}) \wedge \neg(\mathit{coffee} \wedge \mathit{tea})$. It could be considered more transparent to have two intention generation rules, i.e., one for getting coffee and one for getting tea, and having the semantics of goal adoption rules take care of the agent not getting tea *and* coffee. Also, these intention generation rules would be more reusable, since they could, e.g., also be used if the agent has, in a different setting, the goal to have tea *and* coffee. Both rules could then be applied sequentially, resulting in the agent getting both tea and coffee (provided the programmer has specified appropriate plans).

We now return to the problem as explained in Example 4.4 above. This problem can be overcome by slightly adapting the semantics of goal formulas. The idea is that existing intentions should provide a screen of admissibility, i.e., a new intention should be compatible with already existing intentions. This can be realized by taking the goals of the existing intentions, i.e., the second element of each intention tuple, as "facts" of the default theory resulting from the goal adoption rules.

Definition 4.15 *(semantics of goals (alternative))* Let $\langle \sigma, \gamma, \iota, \mathcal{R} \rangle$ be a configuration, let $\mathcal{R} = \langle \mathsf{IG}, \mathsf{GA} \rangle$, and let $\delta = \{\phi \mid \langle \pi, \phi \rangle \in \iota\}$. Let GA' also be as in Definition 4.12, i.e., as follows:

$$\{\kappa \Rightarrow^{+}_{\mathbf{G}} \phi \mid \exists(\beta, \kappa \Rightarrow^{+}_{\mathbf{G}} \phi) \in \mathsf{GA} : \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_{\mathcal{L}_{\mathsf{B}}} \beta\}.$$

The alternative semantics of goals is then as in Definition 4.12, except that E is an extension of $\langle \delta, f(g(\gamma)) \cup f(\mathsf{GA}') \rangle$.

Instead of taking an empty set as the first element of the default theory, we thus use δ. This results in the goals of an agent configuration always being a superset of the intentions of a configuration.[13] This is in line with work on BDI logic [Rao and Georgeff, 1991], which could be considered an additional advantage. The semantics of intention generation can then be defined as follows, where $\models^{E}_{d}$ is the semantics of goals of Definition 4.15.

Definition 4.16 *(semantics of intention generation (alternative))* Let $c = \langle \sigma, \gamma, \iota, \mathcal{R} \rangle$ be a configuration, let $\mathcal{R} = \langle \mathsf{IG}, \mathsf{GA} \rangle$ and let $\beta, \kappa \Rightarrow_{\mathbf{I}} \langle \pi, \phi \rangle \in \mathsf{IG}$ be an intention generation rule. The semantics of applying this rule is then as

[12] Admittedly, a very simple plan to achieve this goal is a plan to get tea. It is however questionable whether a plan for getting tea would do justice to the idea that the agent should get either tea or coffee. Alternatively, one could design a plan which chooses randomly to get either tea or coffee. Such a plan would however have to be adapted if one wants to specify that, e.g., the agent should get tea, coffee, or lemonade.

[13] That is, of the declarative parts of the intentions of a configuration.

follows, where $\iota' = \iota \cup \{\langle \pi, \phi \rangle\}$.

$$\frac{c \models_{\mathcal{L}_{\mathsf{B}}} \beta \qquad \exists E : c \models_d^E \kappa}{\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \rightarrow \langle \sigma, \gamma, \iota', \mathcal{R} \rangle}$$

The third condition of the antecedent of the transition rule of Definition 4.14 has thus been removed, since the function of screen of admissibility of the intentions is now embedded in the semantics of goals. Revisiting Example 4.4, we can see that if the agent has *coffee* as an intention, the default rule corresponding with the goal of having tea is not applicable, since *coffee* is now part of every extension. The goal of having tea can thus no longer be derived, and therefore the second intention generation rule cannot be applied.

In the case of the tea and coffee example, negative goal literals are used to express that the *goals* tea and coffee are incompatible. Given the semantics of intention generation as defined above, tea and coffee are consequently not pursued simultaneously (which is as desired). Negative goal literals can, however, also be used to prevent two goals from being pursued simultaneously, because the *plans* for achieving these goals interfere with each other, for example because they need the same resource. The goal adoption rules will then have to be designed together with the intention generation rules, and it is up to the programmer to identify possible resource conflicts in the plans for certain goals. This could be considered an alternative to explicitly specifying the resources needed by a plan, and having the agent reason about possible resource conflicts, as proposed by Thangarajah et al. [Thangarajah et al., 2002]. Nevertheless, if conflicts among plans are modeled by specifying that the goals of these plans conflict, one does not take into account the fact that a certain plan for a goal ϕ might conflict with a plan for goal ϕ', while another plan for goal ϕ does *not* conflict with the plans for ϕ'.

Discussion and Future Research

Although the above semantics of intention generation is reasonably satisfactory, there are a number of issues related to the dynamics of goals and intentions which it does not address. The first issue we discuss is related to goals which are derived on the basis of other goals, and how this is related to intention generation.

Example 4.5 *(chaining)* Consider an agent with the following goal adoption rules and intention generation rules.

$$\begin{array}{lll} \top & \Rightarrow^{+}_{\mathbf{G}} & p_1 \\ \mathbf{G}p_1 & \Rightarrow^{+}_{\mathbf{G}} & p_2 \\ \mathbf{G}p_1 & \Rightarrow_{\mathbf{I}} & \langle \pi_1, p_1 \rangle \\ \mathbf{G}p_2 & \Rightarrow_{\mathbf{I}} & \langle \pi_2, p_2 \rangle \end{array}$$

The default rules corresponding with the goal adoption rules have one extension, i.e., $\{p_1, p_2\}$. △

Assume the agent of Example 4.5 applies the second intention generation rule, yielding the intention base $\{\langle \pi_2, p_2 \rangle\}$. The intention to achieve p_2 is thus generated, because the agent has a goal p_1. The question is now whether the agent should also try to achieve p_1, i.e., adopt p_1 as intention, since p_2 could be derived as a goal only because p_1 was derived as a goal. One way to view this issue, can be illustrated using the cargo example.

Example 4.6 *(carrying cargo (continued))* Consider the agent of Example 4.2. We repeat the goal adoption rules that are relevant to the current example, and we add two intention generation rules, i.e., one for getting from the waypoint to the target, and one for getting from the source to the waypoint.

$$\begin{array}{lll} \mathbf{B}(\mathit{source} \wedge \mathit{haveCargo}) & \Rightarrow^{+}_{\mathbf{G}} & \mathit{target} \\ \mathbf{B}(\mathit{source}), \mathbf{G}(\mathit{target}) & \Rightarrow^{+}_{\mathbf{G}} & \mathit{waypoint} \\ \mathbf{B}(\mathit{waypoint}), \mathbf{G}(\mathit{target}) & \Rightarrow_{\mathbf{I}} & \langle \mathit{goToTarget}, \mathit{target} \rangle \\ \mathbf{B}(\mathit{source}), \mathbf{G}(\mathit{waypoint}) & \Rightarrow_{\mathbf{I}} & \langle \mathit{goToWaypoint}, \mathit{waypoint} \rangle \end{array}$$

Assume the agent believes he is at the source, and has cargo. The extension of the default rules corresponding with the goal adoption rules with a true belief condition, is then $\{\mathit{target}, \mathit{waypoint}\}$. The first intention generation rule cannot be applied, since the belief condition is false. The intention base resulting from application of the second intention generation rule is $\{\langle \mathit{goToWaypoint}, \mathit{waypoint} \rangle\}$. △

These rules have the same structure as the rules of Example 4.5 (with the addition of some belief conditions), i.e., $p_1 = \mathit{target}$, and $p_2 = \mathit{waypoint}$. In Example 4.6, the goal to go to the waypoint, which was selected as intention, was derived as a goal because the agent had the goal of getting to the target. We now ask again the question of whether the agent should also try to achieve to be at the target. In this example, the answer seems to be *yes*, since it does not make sense to go to the waypoint, without trying to get to the target afterwards. The effort of going to the waypoint would in that case have been in vain.

This example thus seems to suggest that in case goal adoption rules model the adoption of landmarks, it should be the case that if the agent selects a goal as an intention which was derived on the basis of another goal, then the latter goal should also become an intention as soon as possible. The semantics of intention generation of Definition 4.16, however, does not incorporate this. Addressing this issue is left for future research.[14] Nevertheless, there might still be other situations in which case it is not necessary to have such a mechanism.

Another issue is the dual of the issue we just discussed. Consider again Example 4.5, and assume that the agent applies the first intention generation rule, instead of the second, i.e., p_1 becomes an intention. One can now ask the

[14] One way of addressing it could be to make sure that the agent always applies intention generation rules, if they are applicable. A more in-depth analysis is however needed to analyze the exact properties of such a mechanism.

question of whether the agent should also try to achieve p_2, i.e., a goal which can be derived on the basis of p_1.

One situation in which this can be desirable, is if the rules model some kind of obligation. For example, assume that p_1 represents the goal of obtaining an item in a store, and that p_2 represents paying for the item. One would then want the rules to express that if p_1 is pursued (and achieved), p_2 also *has* to be pursued. In other cases, one might want to interpret the rules differently, in the sense that if p_1 is pursued, p_2 *may* also be pursued. One could say that the latter case is realized by the semantics of Definition 4.16. The former case is however not addressed, and is left for future research.

Finally, we remark that Definition 4.16 does not prevent the derivation of two intentions for the same goal. This is partly prevented by the fact that the intention base is a set, which thus does not contain duplicates. Once a part of the plan of an intention is executed, however, the agent might generate the original intention again by applying the relevant intention generation rule again. Since the plan component of the former intention has now changed, duplicates like this are not filtered out. This problem can be remedied in a relatively simple way, i.e., by requiring that an intention generation rule with consequent $\langle \pi, \phi \rangle$ can only be applied if the intention base does not already contain an intention for goal ϕ. More involved definitions might also prevent the adoption of, e.g., an intention for $p \wedge q$, if the intention base already contains an intention for p and an intention for q.

4.5 Related Work

The idea of using default logic to define the semantics of goal adoption rules was inspired by the BOID framework [Broersen et al., 2002] [Dastani and van der Torre, 2004], which uses default logic for generating goals and other mental attitudes. This framework was in turn inspired by Thomason [Thomason, 2000], who uses default logic to develop a formalism to integrate reasoning about desires with planning, and Horty [Horty, 1994], who showed how obligations can be formalized using default logic, and how this formalization is related with the work on deontic reasoning by Van Fraassen [van Fraassen, 1973]. In this section, we compare these approaches to our work.

First, we discuss the work of Horty (Section 4.5.1), and then we address the BOID framework, and the work of Thomason, and of Governatori and Rotolo [Governatori and Rotolo, 2004], which are related to BOID (Section 4.5.2).

4.5.1 Van Fraassen and Horty

In this section, we discuss the relation of our work with work on deontic logic by Horty [Horty, 1993, Horty, 1994, Horty, 1997], and with the work of Van Fraassen [van Fraassen, 1973], which Horty addresses in his work. Our work as

presented in this chapter has been developed independently from the work of Horty. It however turns out that some of it is closely related with his work.

Deontic logics are logics for describing normative reasoning. Since its inception in the work of Von Wright [von Wright, 1951], deontic logic has been developed primarily as a species of modal logic. In [Horty, 1993], Horty however argues that these modal deontic logics do not allow *normative conflicts*. He argues that normative conflicts occur often in everyday life, and that it is thus important that deontic logics are designed that can be used to represent and reason with these conflicts.

A situation gives rise to a normative conflict, if two conflicting propositions can both be said to be obligatory in that situation, i.e., if both $\bigcirc\phi$ and $\bigcirc\neg\phi$ hold for some proposition ϕ. In a basic modal logic K (see, e.g., [Meyer and van der Hoek, 1995]), this would imply $\bigcirc(\phi \wedge \neg\phi)$ and therefore $\bigcirc\bot$. In standard deontic logic, besides the axiom **K**, also the axiom **D**, i.e., $\neg(\bigcirc\phi \wedge \bigcirc\neg\phi)$, is adopted. By adopting this axiom, standard deontic logic thus rules out normative conflicts [Horty, 1993].

In [Horty, 1993] and the follow-up papers [Horty, 1994, Horty, 1997], Horty discusses an approach to reasoning in the presence of normative conflicts which was first proposed by Van Fraassen [van Fraassen, 1973]. The latter paper contains two suggestions, where the second is a refinement of the first. Departing from modal logic and its possible world semantics, Van Fraassen defines obligations on the basis of a set of so-called background imperatives. These background imperatives are essentially propositional formulas, and are supposed to represent the (possibly conflicting) obligations as arising from various sources.

Van Fraassen's initial suggestion is to define the obligations that can be derived from a set of background imperatives γ, as follows:

$$\gamma \models_{F1} \bigcirc\phi \Leftrightarrow_{def} \exists\phi' \in \gamma : \phi' \models \phi.^{15}$$

Comparing this definition to Hindriks' definition for the semantics of goals (Definition 4.7), we can see that it is completely analogous. That is, with the exception that Hindriks requires each individual goal in γ to be consistent. Van Fraassen's definition will allow the derivation of any obligation if there is an inconsistent obligation in the set of background imperatives, while Hindriks prevents this by requiring that each goal in the goal base is consistent. The motivations provided by both authors for their definitions are also very similar: ϕ is a goal or obligation if it is a necessary condition for fulfilling a goal or obligation in the goal base or set of background imperatives, respectively.

As noted by Horty [Horty, 1994], this initial suggestion runs into difficulties, however, when it comes to logical interconnections among imperatives. The example provided by Van Fraassen and Horty to illustrate these difficulties, is the following. Suppose that $\gamma = \{p \vee q, \neg p\}$ is the set of background imperatives.

[15] We rephrase the definition as given in [Horty, 1994] for reasons of comparison, which in turn rephrases the definition given in [van Fraassen, 1973].

Intuitively, one would want to conclude from this that $\bigcirc q$. This however does not follow under Van Fraassen's initial definition, as there is no single imperative from which q follows. To remedy this problem, Van Fraassen provides another and somewhat involved model theoretic definition, which we will refer to using $\models_{F2}$. We do not repeat that definition here, since this would require the introduction of a number of auxiliary notions, and the definition itself is not important for the current discussion.

What *is* important, is that Horty provides an equivalent definition by translating the set of background imperatives of Van Fraassen into default rules [Horty, 1993]. To be more specific, each formula ϕ in the set of background imperatives γ is translated into a default rule $\top : \phi/\phi$. Horty then shows the following, where D_γ is the resulting set of default rules:

$$\gamma \models_{F2} \bigcirc\phi \Leftrightarrow \exists E : E \text{ is an extension of } \langle \emptyset, D_\gamma \rangle \text{ and } E \models \phi.$$

Even though Horty does not provide his own definition of the semantics of obligation, he could have suggested the following as a definition[16], which we will use for comparing his work with ours.[17]

$$\gamma \models_H \bigcirc\phi \Leftrightarrow_{\text{def}} \exists E : E \text{ is an extension of } \langle \emptyset, D_\gamma \rangle \text{ and } E \models \phi$$

If the set of goal adoption rules of $\mathcal{R}$ is empty, we have that $\models_H$ is related to our default semantics as expressed in the following proposition.

Proposition 4.7 Let $\mathcal{R} = \langle \mathsf{IG}, \emptyset \rangle$. Then the following holds.

$$\gamma \models_H \bigcirc\phi \Leftrightarrow \exists E : \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_d^E \mathbf{G}\phi$$

Proof: We have that $\langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_d^E \mathbf{G}\phi$ iff $E \models \phi$, where E is an extension of the default theory $\langle \emptyset, f(g(\gamma)) \rangle$, which is equal to $\langle \emptyset, D_\gamma \rangle$. □

We thus have that the default semantics is related to $\models_H$ in the same way that the default semantics is related to the consistent subset semantics, as expressed in Theorem 4.2. That is, even though $\models_H$ and $\models_d^E$ are based on the same set of default rules, i.e., $D_\gamma = f(g(\gamma))$, we can see that they are not equivalent in the strict sense, because the existential quantifier is not embedded in the semantic definition of $\models_d^E$ (see also the discussion below Theorem 4.2).

[16] Note that we use $\models_H$ to refer to Horty's definition, and $\models_h$ to denote Hindriks' definition (see Definition 4.7). Also, Horty actually uses an entailment relation $\vdash_F$ rather than a satisfaction relation when referring to Van Fraassen's account of deontic logic. In [van Fraassen, 1973], Van Fraassen however says to propose a *truth-definition* of $\bigcirc\phi$, rather than specifying an entailment relation. Since we also propose truth-definitions of goal formulas in this chapter, this is what we use when discussing Van Fraassen's and Horty's work in this section.

[17] Note that the following *defines* $\models_H$, as indicated by $\Leftrightarrow_{\text{def}}$, whereas above we have repeated a *proposition* of Horty regarding $\models_{F2}$, as indicated by $\Leftrightarrow$. In [van Fraassen, 1973], $\models_{F2}$ is not defined in terms of default logic.

The fact that we have Proposition 4.7 is actually not surprising, since Horty shows in [Horty, 1993] that $\models_H$ is equivalent with the consistent subset semantics (to which he does not refer in these terms)[18]. That is, he proves that the consistent subset semantics is equivalent with Van Fraassen's second suggestion, which he has in turn proven to be equivalent to $\models_H$ [Horty, 1993].

We summarize these results below, where $\Leftrightarrow$ denotes equivalence, and $\leftrightsquigarrow$ denotes "equivalence" in the sense of Theorem 4.2 and Proposition 4.7.

$\models_s$	$\Leftrightarrow$	$\models_{F2}$	$\Leftrightarrow$	$\models_H$	[Horty, 1993]
$\models_s$	$\leftrightsquigarrow$	$\models_d^E$			Theorem 4.2
$\models_H$	$\leftrightsquigarrow$	$\models_d^E$			Proposition 4.7

The result of Proposition 4.7 could thus have been obtained by combining Horty's results to conclude that $\models_H$ is equivalent with $\models_s$, and then using Theorem 4.2. Alternatively, Theorem 4.2 could have been obtained by using Proposition 4.7 and the work of Horty.

Comparison with KD

Horty compares in his papers the second definition of Van Fraassen with the modal logics **KD** and **EM**. The logic **EM** is weaker than **K**. Here, we repeat Horty's results with respect to the relation between **KD** and $\models_{F2}$. Since $\models_{F2}$ and the consistent subset semantics are equivalent, we can also relate that semantics to **KD**. Further, we address the relation between the default semantics for goals, and **KD**.

Horty shows that if the set of background imperatives is *consistent*, the logic **KD** and Van Fraassen's second proposal are equivalent. That is, if the set of background imperatives γ is "modalized", i.e., transformed into $\gamma' = \{\bigcirc\phi \mid \phi \in \gamma\}$, we have that $\gamma \models_{F2} \bigcirc\phi$ iff $\gamma' \models_{KD} \bigcirc\phi$, where ϕ is a propositional formula not including any $\bigcirc$ operators. This is in line with Propositions 4.1 and 4.2, in which we show that if the goal base is consistent, the consistent subset semantics is equivalent with the basic semantics, which satisfies **K** and **D**.

Also, Horty shows that if $\bigcirc\phi$ can be derived under Van Fraassen's second proposal, it can be derived under **KD**. As just mentioned, if the set of background imperatives is consistent, the implication also holds in the other direction. However, if the set of background imperatives is *inconsistent*, anything can be derived under **KD**, but not under $\models_{F2}$. In that case, the implication thus does not hold in the other direction.

For our default semantics, we have that the axioms **K** and **D** are satisfied (under a given extension), even if the goal base is inconsistent (Proposition 4.4). This however does not mean that the default semantics and **KD** are equivalent,

[18]The "consistent subset semantics" is not provided by Horty as an independent proposal to define the semantics of obligation, but it is used to simplify the proofs of some other theorems.

in the sense that $\exists E : \langle \sigma, \gamma, \iota, \mathcal{R} \rangle \models_d^E \mathbf{G}\phi$ (where $\mathcal{R}$ contains an empty set of goal adoption rules), iff $\gamma' \models_{KD} \bigcirc\phi$ (where γ' is the modalized version of γ, as above). If γ is consistent, this *does* hold. This can be concluded using Theorem 4.2, and using that in case γ is consistent, **KD** and $\models_{F2}$ are equivalent, and $\models_{F2}$ is equivalent with $\models_s$. If however γ is inconsistent, anything can be derived from **KD**, while this is not the case for the default semantics.

The axioms **K** and **D** are thus satisfied by $\models_d^E$, without being able to derive anything under $\models_d^E$ if the goal base is inconsistent. This stems from the fact that $\models_d^E$ is parameterized by a (consistent) extension, and Proposition 4.4 establishes that **K** and **D** are satisfied for the goals *within* this extension. This is not surprising, since extensions are consistent.

Discussion

While the proposals by Van Fraassen and Horty in the context of deontic logic are thus closely related with our work, we have in the above not addressed our proposal for *goal adoption rules*, and the corresponding semantics of goals in terms of default logic. In the comparisons, we have assumed that the set of goal adoption rules is empty. Consequently, we have only considered default rules of the form $\top : \phi/\phi$, which result from a translation of the goal base into default rules, and which are also the only kind of default rules which Horty considers. In our translation of goal adoption rules to default rules, we use by contrast the *full* power of default rules. To the best of our knowledge, there are no proposals that use default rules in a similar way in the agent programming and deontic logic literature.

These considerations raise the question of how obligations and goals are related. In this chapter, we have not discussed in detail what exactly a goal is, other than that it describes a state that the agent wants to achieve. One way to discriminate goals and obligations on a conceptual level, is by viewing the former as *internal motivational attitudes*, while considering the latter as *external*. The fact that similar proposals have been developed in the field of deontic logic to formalize obligations and in cognitive agent programming to formalize goals, then on the one hand seems reasonable since both obligations and goals are motivational attitudes. Also, dealing with conflicts is an issue both in the case of obligations and in the case of goals, so it is perhaps not surprising that similar mechanisms can be used to handle these conflicts in both cases.

On the other hand, one could argue that the goals of an agent as *internal* motivational attitudes, will *have* to differ from obligations as *external* motivational attitudes, i.e., one could argue that obligations and goals are inherently different notions. Under this assumption, we could conclude that any proposal for defining goals and obligations which does not distinguish between the two mental attitudes, is an oversimplification.

One aspect on which internal and external motivational attitudes might differ, is suggested by the BOID framework, which will be discussed in Section

4.5.2. That framework incorporates both desires, which can be viewed as internal motivational attitudes, and obligations.[19] Desires differ from obligations with respect to *prioritization*, i.e., an agent might give a higher priority to the fulfillment of obligations than to the fulfillment of his own desires, or the other way around. Nevertheless, the *structures* by means of which obligations and desires are represented is the same in that framework.

In our view, the question of how obligations and goals are related in general is still open, and a careful and extensive study of the literature on both topics might shed some light on the issue. Intuitively, we would conjecture that there are differences between goals and obligations in general. However, we also believe that it depends on the context which aspects of goals and obligations one would want to model. If certain aspects of the two notions are not relevant in some context, the notions might coincide. This seems to be the case when considering conflicts among goals and obligations, as discussed above.

4.5.2 BOID and Related Approaches

The idea of using default logic to define the semantics of goal adoption rules was taken from the BOID framework [Broersen et al., 2002] [Dastani and van der Torre, 2004]. This framework was in turn inspired by work by Thomason [Thomason, 2000]. In this section, we first briefly address the work by Thomason, and then we discuss two variants of the BOID framework, i.e., the version as presented in 2002 [Broersen et al., 2002] and the version presented in 2004 [Dastani and van der Torre, 2004]. We will refer to the first variant as BOID'02 and to the second as BOID'04. Finally, we discuss related work in the area of defeasible logic by Governatori and Rotolo [Governatori and Rotolo, 2004], which is also in some sense related to the BOID framework.

Thomason

In [Thomason, 2000], Thomason proposes to model beliefs and desires using normal default rules. He distinguishes what he calls belief-based and desire-based default rules, for representing belief and desires, respectively. A belief-based default rule $\phi : \psi/\psi$ is represented as $\phi \overset{B}{\hookrightarrow} \psi$, and a desire-based rule similarly as $\phi \overset{D}{\hookrightarrow} \psi$. Intuitively, belief-based rules are used for deriving beliefs, and desire-based rules for deriving desires or goals. The antecedent of these rules can refer either to desires or to beliefs.

The way in which belief-based and desire-based defaults are used, is the following. Both types of rules can be used for generating a single extension, i.e., what is derived using a belief-based rule can be used as "input" for a

[19] In BOID, goals are derivative, i.e., goals are generated on the basis of desires and obligations (and beliefs and intentions).

desire-based rule, and vice versa. This means, that formulas derived using belief-based as well as those derived using desire-based defaults, become part of one and the same extension. The formulas of an extension which were derived using desire-based defaults, become the goals (or "wants", in the terminology of Thomason) of this extension. Since extensions are consistent, we know that these goals are not conflicting. When calculating an extension, belief-based rules take precedence over desire-based rules, in order to prevent the agent from doing wishful thinking, i.e., adopting something as a goal which the agent already believes to be unachievable.

We point out a number of differences between the work of Thomason and our work. First, Thomason does not introduce a logical language of goals or desires, like we do, and he consequently does not carry out a semantic analysis of such a language. He does not define what it means that an agent has a goal, or does not have a goal, given some set of belief-based and desire-based defaults. Our logics of goals provide the means for expressing that a formula *is* a goal, and also that a formula is *not* a goal. This kind of expressivity is exploited in the goal adoption rules, which can be used to express that two goals are incompatible (even though logically consistent) by using negative goal literals. Also, our logics of goals allow us to compare properties of our goal operator with properties of operators from modal logics, as we have done in Sections 4.2.2 and 4.3.2.

Another important difference is that he does not separate the derivation of beliefs, from the derivation of goals. This results in beliefs and goals becoming part of one and the same extension. This differs from our agent programming framework, in which we have a belief base for representing beliefs, and a goal base and goal adoption rules for representing goals. Finally, we remark that where Horty considers supernormal defaults, and we consider defaults in general, Thomason uses normal defaults. He does not elaborate on why he chooses to use normal defaults.

BOID'02

In [Broersen et al., 2002], BOID'02 is presented as an architecture for goal generation based on the conditional mental attitudes of beliefs, obligations, intentions, and desires. It builds on the work by Thomason as discussed in the previous paragraph by adding obligations and intentions. As in Thomason, all four mental attitudes are represented using conditional rules,[20] which are interpreted as normal default rules. Also, as in the work of Thomason, some mental attitudes may take precedence over others.

Departing from Thomason however, this prioritization is not fixed. Broersen et al. show that different choices with respect to this prioritization, result in

[20]Note that Thomason uses the terms belief-based and desire-based *default rules* to refer to the conditional rules, while in BOID'02 the rules are referred to as beliefs, desires, obligations, and intentions.

different agent types. For example, if intentions take precedence over desires and obligations, the authors say that the agent is stable. Also, if desires take precedence over obligations, the agent is called selfish, and if it is the other way around, the agent is social, etc.

The differences we pointed out between the work of Thomason and our work, also apply to BOID'02. The distinction made in BOID'02 between obligations, desires, and intentions, which essentially imposes a prioritization on the default rules, could be added to our framework by introducing such an ordering on the goal adoption rules, and the rules resulting from the goals in the goal base. However, as we sketched in Section 4.1.1, we propose to use our goal generation mechanism in the context of cognitive agents with a separate intention base. Intentions in our proposal thus form a separate component of the agent, with a different structure and semantics than goal adoption rules and the corresponding default rules.

BOID'04

In BOID'04 [Dastani and van der Torre, 2004], an agent specification contains belief rules, obligation rules, intention rules, and desire rules, as in BOID'02. The difference is however, that the antecedent of these rules can be a modal logic formula, containing an arbitrary nesting of **B**, **O**, **I**, and **D** operators. Also, the consequent of a belief rule should be a formula of the form $\mathbf{B}(\psi)$, where ψ is an arbitrary modal logic formula. The consequent of an obligation, intention, and desire rule, should similarly be of the form $\mathbf{O}(\psi)$, $\mathbf{I}(\psi)$, and $\mathbf{D}(\psi)$, respectively. This thus differs from BOID'02, in which both the antecedent and the consequent of the rules are propositional formulas, representing beliefs or goals of the agent.

The way in which these rules are used, is comparable with the way in which normal default rules are used, i.e., if the antecedent follows from the set of formulas generated so far, and the consequent is not inconsistent with this set, then the formula in the consequent can be added to this set. The authors assume some modal logic consequence relation for establishing whether the antecedent follows from the set of formulas generated so far. Instead of using standard (propositional) default logic to generate propositional formulas representing goals as in BOID'02, the framework of BOID'04 thus generates sets of modal formulas, representing beliefs, obligations, intentions, and desires.

We point out a number of differences between our work and the work by Dastani and Van der Torre on BOID'04. Most importantly, like Thomason and BOID'02, BOID'04 does not aim to investigate semantics of a logical language of goals, like we do. They do use **B**, **O**, **I**, and **D** operators in their language, but the semantics of these operators is stated to be that of standard *modal* operators.

Another difference between our work and BOID'04, is that we use standard propositional default logic to interpret our rules, while BOID'04 defines its own

procedures for generating extensions, in which a modal logic consequence relation is assumed. Further, in BOID'04, the extensions which are computed are explicitly stored in the agent configurations, including the extension which is chosen to form the agent's intention base. Extensions are computed anew, if an appropriate meta-level action is executed. We do not compute a set of extensions and store these. Rather, we take the intention generation rules as the basis, and in order to decide whether such a rule can be applied, we have to check whether the goal formula in its antecedent holds. In order to check this, the extensions have to be computed. These extensions are computed when the agent wants to apply an intention generation rule, and we do not store the computed extensions.

An advantage of selecting a single extension and storing this as the intention base, is that the agent will for sure pursue a compatible set of goals. It might however be the case that the agent does not have a plan for some of the intentions. This problem is avoided when taking the intention generation rules as the basis. In that case however, one needs to take care that the intention base keeps consisting of compatible intentions, and a number of other issues need to be addressed, as discussed in Section 4.4.2.

Finally, we remark that due to the interpretation of rules in BOID'04, it can be the case that a rule is applied which has, e.g., $\neg\mathbf{D}(\psi)$ as its antecedent, while a rule which is applied later on in the process, adds the formula $\mathbf{D}(\psi)$ to the extension which is being generated. It can thus be the case that a formula is added to an extension because the agent does not desire ψ in that extension, while later on precisely this desire of ψ is added to that same extension through the application of another rule. We prevent this by translating negative goal formulas in the antecedent of a goal generation rule to the justification of the corresponding default rule, as explained in Section 4.3.1.

Governatori and Rotolo

Governatori and Rotolo [Governatori and Rotolo, 2004] build on work on defeasible logic by Nute [Nute, 1994] (see also [Dastani et al., 2005a] for a follow-up paper). A theory in defeasible logic may contain three kinds of rules, i.e., strict rules for deriving indisputable facts, defeasible rules for deriving defeasible conclusions, and defeaters, which can be used to block the derivation of certain conclusions. Also, a defeasible theory contains a set of indisputable facts, and a superiority relation, which expresses precedences among rules. The antecedent of the different kinds of rules is a set of literals, and the consequent is a literal.

Governatori and Rotolo extend defeasible theories as proposed by Nute, by distinguishing between rules for deriving knowledge, intentions, so-called agency, i.e., intentional actions, and obligations. The idea is that, e.g., an obligation rule can be used to derive a literal expressing an obligation, and similarly for the other kinds of rules. Further, the literals in the antecedent of a rule can be modalized, i.e., a defeasible obligation rule $\mathbf{I}(p) \Rightarrow_{\mathbf{O}} q$, for example, expresses

that the obligation q can be defeasibly derived, if p can be defeasibly derived on the basis of an intention rule. As explained by Governatori and Rotolo, the rules are thus used to devise logical conditions for introducing modalities, i.e., if q can be derived on the basis of an obligation rule, the agent has an obligation for q.

Our work differs from that of Governatori and Rotolo in several ways. Firstly, our work builds on default logic, while theirs builds on defeasible logic. The latter has a skeptical semantics, i.e., something can only be derived if there is no information to the contrary, while the former can be used with a credulous or a skeptical interpretation, possibly giving rise to multiple extensions of a theory. Consequently, in our framework we have had to address how to handle multiple extensions. Also, although default logic facilitates a skeptical as well as a credulous interpretation, we propose to use the credulous approach in the application of intention generation rules for checking whether the agent has a goal. This is based on the intuition that the agent can choose to try to reach any of the goals it has, even if it conflicts with another goal of his. It can just not fulfill these at the same time.

Secondly, we allow negated goal formulas in the antecedent of goal adoption rules, and provide an appropriate translation of these into the framework of default logic. Governatori and Rotolo on the other hand do not allow this. Nevertheless, it could perhaps be added to their framework by introducing a construct in the antecedent of rules to express explicit failure as suggested in [Maher and Governatori, 1999], comparable with negation as failure in logic programming.

Also, since Governatori and Rotolo build on defeasible logic, the formulas which can be derived in their logic are tagged (modalized) literals. Our approach on the other hand provides various semantics of a logical language of goals that includes negation and conjunction. As explained, this provides for added expressivity in goal adoption rules, and it allows us to compare properties of our goal operator with properties of operators from modal logics. Moreover, Governatori and Rotolo *extend* defeasible logic. We do not extend default logic, but *translate* goal adoption rules into default rules, and define the semantics of goal formulas based on the extensions of the resulting default theory. Generally speaking, it can be beneficial to work within an existing framework rather than extending one, since in the former case one can reuse all knowledge that has been established with respect to the framework.

Further, the focus in the work of Governatori and Rotolo is on the interplay between the various mental attitudes, while in our approach we only consider goals.[21] In the former, a certain mental attitude can be derived on the basis of another, and the derivation of one mental attitude can block the derivation of others. It is not immediately clear whether, and if so, how, our work can be extended to incorporate that kind of reasoning. Since we transform goal

[21] And beliefs, but beliefs play a relatively simple role in defining the semantics of goals.

adoption rules to propositional default rules, we loose the explicit representation of mental attitudes. In this sense, our work is thus more closely related to BOID'02, since in the latter also the derivation of only goals is considered, albeit on the basis of different kinds of rules.

4.6 Conclusion

We have explored semantics of declarative goals in an agent programming setting, where goals may conflict. We have investigated semantics based on the goal base of the agent, and examined their properties and interrelations. These semantics turn out to be closely related to work in the area of deontic logic by Horty and by Van Fraassen. Further, we have proposed a semantics for goals based on the goal base and a set of goal adoption rules. This semantics was defined by translating the goal adoption rules into default rules. We have argued that the translation gives intuitive results, we have investigated properties of this semantics, and have shown that it is closely related to the consistent subset semantics that does not take into account goal adoption rules.

Also, we have compared our proposal with other research on the representation of goals and desires, that builds on default logic. The main difference between our work and those proposals is that, in contrast with those, we define and investigate various semantics of a logical language of goals, and we use the full power of default logic to define our semantics. This gives us added expressivity, and allows us to compare properties of our goal operator with properties of modal operators. Moreover, we have suggested how our semantics could be embedded in a cognitive agent programming framework with intention generation rules, and pointed out a number of unresolved issues with respect to this.

Besides the future work as mainly addressed in Section 4.4.2, we aim to investigate whether we can implement our semantics of goals using answer set programming. Answer set programming is a form of declarative programming that is similar in syntax to traditional logic programming and which semantics is related to non-monotonic logics, and default logic in particular.

Concluding, we maintain that a systematic analysis of semantics of declarative goals in agent programming is essential, in order to be able to understand how we can best incorporate these in agent programming languages. This chapter contributes to this effort.

Acknowledgements

We would like to thank Henry Prakken for his helpful comments on a version of the paper [van Riemsdijk et al., 2005b], on which this chapter is based. Also, we would like to thank Jan Broersen, Joris Hulstijn, and Leon van der Torre for discussions on the BOID framework.

Chapter 5

Putting Goals in Perspective

In Chapters 3 and 4 we have investigated certain aspects of the representation and semantics of goals in the context of agent programming languages similar to the one presented in Chapter 2. Work of others that is closely related to our researches has been addressed in the former two chapters. Over the past years, the body of research addressing the incorporation of goals in agent programming frameworks has been growing. Different researchers have developed different perspectives towards this issue, resulting in many different views on what constitutes a goal, and many different ways of representing and using goals in agent programming frameworks.

In this chapter, we aim to provide a broad overview of the various approaches that in some form address the incorporation of goals in agent programming frameworks. One might expect such an overview to start with a clear definition of what a goal is (at least in the context of agent programming). It is however difficult to provide such a definition, because of the many different usages of the term. Nevertheless, the approaches discussed in this chapter have in common that a goal is viewed in one way or another as a motivational attitude. In particular, we do not consider goals as used in Prolog as motivational attitudes.

We do not aim to analyze the various approaches in order to conclude that one approach is somehow better than another. Instead, we want to provide some structure to the area by identifying important strands of research regarding ways in which goals have been represented and used in agent programming frameworks. Each approach may have its strengths and weaknesses, and may be more suitable in one setting than in another. While comparing the merits of various approaches is important, such a detailed analysis is beyond the scope of this chapter. Nevertheless, we do feel that the aspects as to which approaches differ from one another as identified in this chapter, may serve as a basis for such comparative analyses.[1]

[1] There is some existing work investigating whether one approach really differs from another, such as the work described in Chapter 3. Also, there is a number of papers in which it is proven

5.1 What Is a Goal?

Although we will not provide a definition of what a goal is, the aim of this section is nevertheless to try to get a handle on this issue to a certain extent. We do not focus solely on research that literally uses the term "goal", but we also consider the related motivational attitudes of desires and intentions. The reason is, that the distinction between these motivational attitudes is not always clear, and sometimes these notions are used interchangeably. Also, properties that are attributed to a certain motivational attitude in one framework, may be attributed to another motivational attitude in another approach.

In this section, we do not discuss cognitive agent programming frameworks themselves. Rather, we treat research that lies at the basis of many of these frameworks. That is, we briefly discuss motivational attitudes as used by the philosophers Dennett and Bratman (Section 5.1.1), and we discuss properties that have been attributed to motivational attitudes in two influential logics (Section 5.1.2).

5.1.1 Philosophy: Dennett and Bratman

Dennett is a philosopher whose approach to the problem of intentionality has been of significant influence on the field of cognitive agent programming. "Dennett suggests that intentionality is not so much an intrinsic feature of agents, rather, it is more a way of looking at agents. Dennett calls the seeing of agents as intentional beings, or beings that act according to their beliefs and desires, as taking the intentional stance" [Douglas and Saunders, 2003].

The idea of the *intentional stance* is thus that the behavior of rational agents can be predicted by ascribing beliefs and desires to the agent, and by assuming that the agent will tend to act in pursuit of its desires, taking into account its beliefs about the world. Dennett himself formulates it as follows.

> *Here is how it works: first you decide to treat the object whose behavior is to be predicted as a rational agent; then you figure out what beliefs that agent ought to have, given its place in the world and its purpose. Then you figure out what desires it ought to have, on the same considerations, and finally you predict that this rational agent will act to further its goals in the light of its beliefs. A little practical reasoning from the chosen set of beliefs and desires will in most instances yield a decision about what the agent ought to do; that is what you predict the agent will do.* [Dennett, 1987, page 17]

Dennett thus proposes to use the mental attitudes of *beliefs* and *desires* for predicting the behavior of rational agents. The idea of Bratman now is that there

that one agent programming language can be embedded in another [Hindriks et al., 1998, Hindriks et al., 2002]. This work however does not specifically focus on goals.

is another mental attitude, different from beliefs and desires, that is essential for a theory of practical rationality. This is the mental attitude of *intention* [Bratman, 1987]. The philosophy of Bratman has been termed Belief Desire Intention (*BDI*) philosophy.

Like desires, but unlike beliefs, intentions are motivational attitudes, or *pro-attitudes*, in Bratman's terminology. "Pro-attitudes [...] play a motivational role: in concert with our beliefs they can move us to act" [Bratman, 1987]. While both intentions and desires are pro-attitudes, their motivational role is different. That is, where desires *influence* future conduct, in the sense that an agent will be more inclined to act towards achieving its desires, intentions *control* future conduct. If an agent, e.g., desires to go shopping after work, it does not necessarily mean that if it comes home from work it will actually go shopping. It might, e.g., have a stronger desire to watch television. If, however, the agent *intends* to go shopping after work, one would normally expect the agent to go shopping indeed. As a conduct-controlling pro-attitude, intentions thus involve a special *commitment* to action that desires do not.

Intentions thus bring forth a commitment to action, when the time comes to act. Besides this kind of commitment that comes into play at the time of action, there is another dimension of commitment in Bratman's philosophy. This second dimension of commitment is concerned with the role of intentions in the period between their initial formation and their eventual execution. The idea here is that once an intention is formed, an agent is committed to this intention in the sense that having an intention will involve a strong disposition *not to reconsider* it, except if a significant problem presents itself. Intentions thus have a characteristic stability.

5.1.2 Formalizing Motivational Attitudes

One way to go about trying to find out what a goal is, is to study properties that people have attributed to goals. These properties can be attributed through a philosophical analysis, like those put forward by Dennett and Bratman. If one wants to be more precise, however, one can try to formalize goals and their properties, e.g., by designing a logic.

Goals, Desires and Intentions

These logics that formalize properties of goals and related motivational attitudes is what we will focus on in this section. We will mainly be concerned with two logics that have been highly influential in the agent systems field, i.e., the logics presented in [Cohen and Levesque, 1990] and in [Rao and Georgeff, 1991]. Both papers propose modal logics to formalize the BDI philosophy of Bratman, thereby formalizing (various kinds of) goals and related notions. These logics are often referred to as *BDI logics*, although some use this term only for the logic presented in [Rao and Georgeff, 1991]. We assume the reader has some familiarity

with modal logic (see [Blackburn et al., 2001, Meyer and van der Hoek, 1995] for background on modal logic).

Both logics formalize various mental attitudes of agents, including goals. Since the logics are aimed at formalizing BDI philosophy, intention is an important notion in both of them. In [Cohen and Levesque, 1990], intentions are defined in terms of beliefs and goals, which are both primitive in the logic. In [Rao and Georgeff, 1991], beliefs, goals, *and* intentions are primitive. Somewhat surprisingly perhaps, since the logics aim to formalize BDI philosophy, they do not formalize *desires*. The following is stated in [Cohen and Levesque, 1990] and [Rao and Georgeff, 1991] about this.

> *Importantly, we do not include an operator for wanting, since desires need not be consistent. Although desires certainly play an important role in determining goals and intentions, we assume that once an agent has sorted out his possibly inconsistent desires in deciding what he wishes to achieve, the worlds he will be striving for are consistent.* [Cohen and Levesque, 1990, page 231]

> *The role played by attitudes such as beliefs (B), desires (D) (or goals (G)), and intentions (I) in the design of rational agents has been well recognized in the philosophical and AI literature [...].* [Rao and Georgeff, 1991, page 473]

> *Although, in the general case, desires can be inconsistent with one another, we require that goals be consistent. In other words, goals are chosen desires of the agent that are consistent.* [Rao and Georgeff, 1991, page 474]

In both papers, goals are thus viewed as a consistent set of chosen desires, without modeling these underlying desires in the logic. Presumably, the authors thus chose not to model desires, since a modal desire operator that allows inconsistencies among desires does not have very interesting logical properties. That is, in normal modal logics, one can derive anything as a desire from an inconsistent set of desires. From $\mathbf{D}p$ and $\mathbf{D}\neg p$, for example, one can derive $\mathbf{D}q$ and even $\mathbf{D}\bot$, the desire to establish the impossible. Nevertheless, while the logic in [Rao and Georgeff, 1991] considers goals and not desires, the goal operator is replaced with a desire operator in a later paper from the same authors [Rao and Georgeff, 1998] (while consistency of desires is still assumed). The authors, however, do not comment on the reason why.

Properties of Motivational Attitudes

When it comes to properties of motivational attitudes, we distinguish two kinds of properties, i.e., *static* and *dynamic* properties. Static properties are properties of mental attitudes in a single state or world, and dynamic properties describe

how an agent's current mental attitudes relate to its future mental attitudes. Static properties can again be distinguished into properties of a single mental attitude, and properties relating different mental attitudes.

With respect to properties of a single mental attitude, both papers stipulate that goals, and in [Rao and Georgeff, 1991] also intentions, obey the axioms **K** and **D** of modal logic. The axiom **K** expresses that goals are closed under classical logical consequence, which can be formalized as $\mathbf{G}(\phi \rightarrow \psi) \rightarrow (\mathbf{G}\phi \rightarrow \mathbf{G}\psi)$. The axiom **D** expresses that the agent cannot have inconsistent goals, which can be formulated as $\neg\mathbf{G}\bot$. Regarding the relation between goals and intentions, [Rao and Georgeff, 1991] specifies that intentions are a subset of an agent's goals, expressed by the axiom $\mathbf{I}\phi \rightarrow \mathbf{G}\phi$[2]. In [Cohen and Levesque, 1990], intentions are defined as a certain kind of goal.

While these axioms do provide us some information as to the properties of goals and intentions, they do not address an important aspect of these mental attitudes, i.e., their dynamics or behavior over time. In line with Bratman, both papers aim to have agents be *committed* in some way to achieving their intentions. This means that an agent cannot give up its intentions, unless some specific conditions hold. Intentions should thus have a certain level of persistency as time goes by. In [Cohen and Levesque, 1990], the appropriate level of persistency of intentions is realized using the definition of certain kinds of goals.[3] In [Rao and Georgeff, 1991], it is realized by providing axioms expressing properties of the dynamic behavior of intentions, i.e., *without* considering goals. Nevertheless, the properties attributed to intentions in [Rao and Georgeff, 1991] have been taken up in agent programming frameworks in the context of goals (see Section 5.4.2). We discuss below in more detail how the two logics address commitment towards motivational attitudes.

In [Cohen and Levesque, 1990], establishing the appropriate level of persistency of intentions is realized using the definition of a so-called *achievement goal*. An achievement goal in that paper is a goal to achieve a certain state of affairs sometime in the future, and the agent should believe this situation is not realized already. On the basis of achievement goals, the notion of a *persistent goal* is then defined. A persistent goal is a goal that "the agent will not give up until he thinks it has been satisfied, or until he thinks it will never be true" [Cohen and Levesque, 1990].[4] This notion of a persistent goal thus captures a certain level of commitment towards goals, which is called *fanatical* commitment in [Cohen and Levesque, 1990]. On the basis of persistent goals, intentions are defined with the same level of commitment as persistent goals.

[2] That is, where ϕ is a so-called O-formula, i.e., a formula containing no positive occurrences of *inevitable* outside the scope of the modal operators **B**, **G**, and **I**.

[3] Recall that in that paper, only beliefs and goals are primitive in the logic.

[4] Achievement goals and persistent goals do not necessarily obey the **K** axiom, since these goals depend on the agent's beliefs [Cohen and Levesque, 1990]. An agent might, e.g., have achievement goals or persistent goals to realize p and to realize $p \rightarrow q$, but this does not mean it also has the goal to achieve q, since q might already be believed to be achieved.

Another type of commitment is captured by what is called a persistent relativized goal in [Cohen and Levesque, 1990]. An agent that has a goal to achieve ϕ, relativized to ψ, should maintain this goal until it believes it has achieved the goal, or believes it is unachievable, or believes $\neg\psi$ is the case. Intuitively, ψ constitutes the agent's reasons for adopting the goal ϕ. The idea is then that if the reason for adopting the goal becomes false, the agent may drop the goal.

In [Rao and Georgeff, 1991], various levels of commitment which an agent might have towards its intentions are introduced by stipulating axioms expressing how current intentions relate to future intentions. The ways in which an agent might behave with respect to commitment towards its intentions, are termed *commitment strategies* in [Rao and Georgeff, 1991]. The paper introduces three commitment strategies: *blind*, *single minded*, and *open minded*. An agent behaving according to the single minded commitment strategy, corresponds with the fanatically committed agent of [Cohen and Levesque, 1990], i.e., such an agent maintains its intentions until it believes it has achieved them, or believes it is impossible to achieve them. A blindly committed agent maintains its intentions until it believes it has achieved them. This type of commitment is thus even stronger than single minded commitment. An open minded agent maintains its intentions until it believes they are achieved or they are no longer its goals.

5.2 Why Goals in Agent Programming?

In this section, we discuss *why* we think goals are important in (cognitive) agent programming. We will not go into detail with respect to ways in which goals have been implemented in agent programming frameworks. The latter will be addressed in Sections 5.3 and 5.4.

5.2.1 Bridging the Gap

One way to motivate the importance of goals, is by revisiting the idea of agent-oriented programming, as first put forward in [Shoham, 1993]. Shoham proposes to use mental attitudes not only to explain and describe the behavior of rational agents, but also to use mental attitudes as components of the programming language itself.

While Shoham proposes a set of mental attitudes differing somewhat from, although presumably inspired by, those of BDI theory,[5] the idea of agent-oriented programming has been taken up by others as the effort to design programming languages that in some sense implement the ideas of BDI logics (and herewith of BDI philosophy). Most notably, the proposal by Rao of the influential cognitive agent programming language AgentSpeak(L) [Rao, 1996] is motivated by the question of how to relate implemented BDI systems to the BDI

[5] Shoham finds his choice of mental attitudes more basic.

logics. It is important that implemented BDI systems have a strong relation with the BDI logics, since the solid theoretical underpinning that these logics provide will make such systems less likely to suffer from ad-hoc solutions. Also, if such a relation can be established, these logics can be used for specification and verification of implemented BDI agents.

The issue of relating cognitive agent programming languages and BDI logics has come to be known as the issue of *bridging the gap* between theory and practice [Rao, 1996, Hindriks et al., 2001, van Riemsdijk et al., 2003b], and was also briefly mentioned in [Wooldridge and Jennings, 1995], and discussed in [van der Hoek and Wooldridge, 2003]. Rao states that the "holy grail of BDI agent research is to show such a one-to-one correspondence [between the model theory, proof theory, and the abstract interpreter] with a reasonably useful and expressive language".[6]

Returning to the issue of the importance of goals in cognitive agent programming, we argue as follows. Since an important aim of cognitive agent programming languages is to "implement" somehow the ideas of BDI logics, and since goals are important in these logics, goals should be considered an important aspect of cognitive agent programming languages as well. Providing an appropriate implementation of goals is a necessary prerequisite for bridging the gap between logics and programming languages. Nevertheless, it might also be necessary to tune the BDI logics to "match" agent programming languages, in order to be able to bridge the gap.

5.2.2 Programming Proactive Agents

Besides from the perspective of bridging the gap, the issue of the importance of goals in agent programming can also be addressed by considering the agent characteristics as suggested in [Wooldridge and Jennings, 1995]. One of these characteristics is *proactiveness*. Proactive agents are "able to exhibit goal-directed behavior by *taking the initiative* in order to satisfy their design objectives" [Wooldridge, 2002]. The fact that goal-directedness is considered an important property of intelligent agents, seems to suggest that goals are inherently important when considering *programming* agents.

The implications of this desired characteristic of goal-directedness for the practice of programming agents, are however not immediately clear. One could, for example, argue that a standard procedural or object-oriented program already exhibits goal-directed behavior. Such a program is written with a certain desired effect in mind, and this effect could be seen as the goal of the program

[6] While the work of Rao on AgentSpeak(L) was motivated by the issue of showing a one-to-one correspondence between an implemented BDI system and BDI logics, he does not actually provide such a correspondence. In the conclusion of [Rao, 1996], he states that "Bridging the gap between theory and practice [...] has proved elusive". Instead, he suggests an alternative approach, in which he proposes the language AgentSpeak(L) with its semantics as an alternative characterization of BDI agents.

(see also [Wooldridge, 2002]). In this case, the goal is *implicit*, i.e., it is not represented and used explicitly in the program. The fact that a procedural program can be viewed as being proactive, might suggest that we do not need new tools and techniques for programming proactive agents.

In the context of intelligent agents which operate in dynamic and uncertain environments, however, this simple model of goal-directed programming is often not enough. The model assumes, in particular, that the environment does not change while some procedure is executing [Wooldridge, 2002]. If problems or unexpected events occur during the execution of a procedure, the program might throw an exception and terminate (see also [Wooldridge and Ciancarini, 2001]). This works well for many applications, but we need something more if change is the norm and not the exception.

One way of tackling the programming of flexible, proactive agents, is by using an *explicit* representation of goals in the agent program. The explicit representation of goals in agent programs is what we focus on in this chapter. By using an explicit representation, goals can, e.g., be maintained until they are achieved. If a plan to achieve a certain goal fails, the agent can select another plan to try again, thereby increasing its flexibility. In this way, plan failure is decoupled from goal failure [Winikoff et al., 2002]. In [Georgeff et al., 1999], this is stated as follows: "Conventional computer software is "task oriented" rather than "goal oriented"; that is, each task (or subroutine) is executed without any memory of why it is being executed. This means that the system cannot automatically recover from failures [...]." This added flexibility is thus an important practical advantage that can be obtained by using an explicit representation of goals.

5.2.3 Goals as a Modeling Concept

Another way to argue why goals are important in agent programming, is by viewing goals as a useful modeling concept when analyzing and designing a system. If a system is analyzed in terms of goals (and other notions), it will generally be easier to go from design to implementation if the programming language with which the system is implemented, contains explicit constructs for representing goals.

An example of a methodology that uses goals as a modeling concept, is the KAOS methodology [van Lamsweerde and Letier, 2004]. KAOS is a method for requirements engineering. It is argued that "goals are an essential abstraction for eliciting, elaborating, modeling, specifying, analyzing, verifying, negotiating and documenting robust and conflict-free requirements" [van Lamsweerde and Letier, 2004]. By using goals in the modeling process, one can, e.g., properly capture and manage positive and negative interactions among system goals.

Tropos [Bresciani et al., 2004] is another example of a software development methodology that incorporates goals as a modeling concept. Tropos is geared

towards cognitive agent programming languages, and intends to support all analysis and design activities in the software development process. Besides the notion of goal, other concepts relevant in the context of cognitive agent programming are used in the methodology.

Finally, we mention work by Norling concerning human modeling. Norling uses BDI philosophy to develop models of human behavior, and then uses a cognitive agent programming framework for implementing the model. We mention in particular research in which expert human players of the computer game Quake were modeled [Norling, 2003]. The model was developed on the basis of knowledge obtained from expert players. Norling claims that when people are "asked about how they think about a problem, people already have a tendency to explain their actions in terms of what their intentions were, which in turn are explained in terms of their goals and beliefs" [Norling, 2003]. The model was implemented using JACK. The translation into JACK was relatively easy, although the lack of explicit representation of goals in JACK needed a work-around.

5.3 Representation

We can conclude from the discussion in Section 5.2, that the importance of the explicit use of goals in agent programs can be motivated in different ways. This is part of the reason for the emergence of various different ways of representing goals in agent programming frameworks. In this section, we compare these approaches for representing goals with respect to three aspects. These aspects have been chosen because, in our view, these identify important strands of research regarding the representation of goals.

While the representation of goals cannot be considered completely independently from the behavior of goals at run-time, we nevertheless try to focus in this section on representational aspects. In Section 5.4, we address the dynamic behavior of goals over time.

5.3.1 Representing Goals Separately or Not

An important aspect as to which approaches for the representation of goals differ, is whether the agent has a separate component for the representation of goals. Such a component is generally called a *goal base*. An alternative for using a goal base, is using an *event base*. Events can be generated, e.g., because the beliefs of an agent have changed, because a message has been sent to the agent, and also because a goal has been generated. These various kinds of events, including the events related to the generation of a goal, are all stored in the event base.

Event Base

The earliest example of a cognitive agent programming language that is based on the generation and processing of events, is AgentSpeak(L). AgentSpeak(L) is a "textual and simplified version of the language used to program the Procedural Reasoning System" (*PRS*) [Georgeff and Lansky, 1987, Ingrand et al., 1992]. PRS is one of the first BDI agent architectures, and the system as discussed in [Ingrand et al., 1992] also proposes the use of events. An AgentSpeak(L) agent consists of a belief base and a plan library consisting of a set of predefined plans. At run-time, an event base and an intention base are created. The event base is used for storing the generated events, and the intention base is used for storing the plans that are currently being executed. Intentions are adopted in response to the occurrence of a certain event.

A plan in AgentSpeak(L) consists of a triggering condition, which specifies the event for which the plan may be selected, a context, which is a condition on the beliefs of the agent, and a body. The body of a plan consists of a sequence of goals which need to be achieved, and actions which should be executed. The body `a1;a2;!p;a3`, e.g., represents that the agent should first execute the actions `a1` and `a2`, then achieve the goal `p` (represented using an exclamation mark), and then execute the action `a3`. The goal `p` can be viewed as the *subgoal* of this plan. If the agent encounters a subgoal such as `p` in a plan, it generates a goal addition event for this subgoal, which is placed in the event base. The agent then, broadly speaking, suspends the execution of the plan, and tries to find a plan in the plan library that has `!p` as its triggering condition. If such a plan is found, it is executed. Then, the agent continues the execution of the plan from which the goal addition event for `!p` was generated, i.e., it executes `a3`.

In the specification of an AgentSpeak(L) agent, (sub)goals are thus present in the plans of the agent. At run-time, these goals generate goal events, which are added to the event base of the agent. The event base not only contains goal events, but may also contain events generated because of a change in beliefs. Nevertheless, one could view the set of goal events in the event base as the current goals of the agent, which is also the approach taken in [Bordini and Moreira, 2004] (in which the term "desires" is used, rather than "goals").

A language that has an event-based execution model comparable with that of AgentSpeak(L), is the language that comes with the JACK platform [Winikoff, 2005]. JACK is an industrial strength agent platform, that includes a programming language that extends Java [Gosling et al., 2000] by adding language constructs for the specification of agent concepts such as beliefs, events, plans, etc. As in the case of AgentSpeak(L), JACK does not have a separate component for the representation of an agent's goals.

Goal Base

An example of a framework that *does* incorporate such a component for representing an agent's goals, is the language of Chapter 2. Further examples are JAM [Huber, 1999], GOAL [Hindriks et al., 2001], Dribble [van Riemsdijk et al., 2003b], the latest version of 3APL [Dastani et al., 2004, Dastani et al., 2005c], the approach presented in [Dastani et al., 2006], and Jadex [Pokahr et al., 2005b]. JAM is a BDI agent architecture inspired by, among others, the work on PRS. GOAL, Dribble, and 3APL are cognitive agent programming language that come with a formal semantics. In [Dastani et al., 2006], a formalization of various kinds of goals is presented, in the context of cognitive agent programming languages (see Section 5.4.2 for further explanation). Jadex is a reasoning engine implemented in Java that provides an execution environment and an API for developing cognitive agents. Each of these frameworks provide the agent programmer with the possibility to endow the agent with a set of goals at start-up. These goals thus form a separate component of the agent at the time of specification, but also at run-time. The agent selects plans on the basis of its goals (and beliefs).

It has been suggested that it is an advantage of AgentSpeak(L) that the language does *not* introduce an extra component for the representation of goals [Hübner et al., 2006]. However, using a goal base for representing goals provides several potential advantages. In particular, it enables logical reasoning with goals in the case of a logic-based representation of goals (see Sections 5.3.2 and 5.4.1 for a further discussion). Moreover, the separate representation of goals can facilitate experimenting with more involved ways of representing goals, such as representations taking into account possible interactions among goals (see Section 5.3.3). Also, several approaches that use a goal base do not make use of an event base [Huber, 1999, Hindriks et al., 2001, van Riemsdijk et al., 2003b, Dastani et al., 2004, Dastani et al., 2006]. Moreover, it has been shown that AgentSpeak(L) can be embedded in the first version of 3APL [Hindriks et al., 1998, Hindriks et al., 1999b], and 3APL does not make use of an event base.

5.3.2 Logic-Based and Non-Logic-Based Approaches

In the previous section, we have discussed the incorporation of a separate component for the representation of goals as a criterion for distinguishing frameworks for the representation of goals. Another such criterion is whether logic is used in the representation and processing of an agent's goals, or not.

In logic-based frameworks, goals are represented as logical formulas that represent the situations an agent wants to achieve. In most of these approaches, logical reasoning is used for checking whether a goal is achieved, and/or for deriving new goals on the basis of existing goals. Traditionally, many frameworks in which the representation of goals is considered important, are logic-based.

Nevertheless, there are also approaches that do not require logical inference mechanisms for processing goals.

Logic-Based

An example of a logic-based framework for representing goals is the language of Chapter 2. Another example is the latest version of 3APL [Dastani et al., 2004, Dastani et al., 2005c]. In these languages, goals are represented by means of a goal base, which consists of a *set of logical formulas* representing the situations the agent should try to achieve. The idea of representing goals in this way was taken from [van Riemsdijk et al., 2003b], the authors of which took this idea in turn from [Hindriks et al., 2001]. An agent in these languages has rules that specify which plan an agent may execute if it has a certain goal (and certain beliefs). New goals can be derived on the basis of the formulas in the goal base by means of logical reasoning. That is, an agent can, e.g., derive the goal to achieve p from the goal $p \wedge q$.

In [Winikoff et al., 2002], another language is presented that incorporates a logic-based representation of goals. Winikoff et al. define a goal as a *logical formula* representing a desired situation, together with a *set of plans* by means of which the agent can try to achieve this situation. A goal also contains a logical formula specifying the condition under which it should be dropped (see Section 5.4.2).

Another proposal that takes a logic-based approach towards the representation of goals, was presented in [Sardina and Shapiro, 2003]. In that paper, the focus is on *prioritized* goals, i.e., being able to represent that one goal has a higher priority than another. The authors extend the language IndiGolog [Giacomo and Levesque, 1999], which is a language from the Golog family [Giacomo et al., 2000]. Goals are represented as a set of formulas endowed with a total order. These goals, together with their ordering, are used for selecting the most appropriate plan, by reasoning about the results of plan execution.

AgentSpeak(L) could also be categorized as a language that has a logic-based representation of goals, since subgoals as occurring in the plans of AgentSpeak(L) agents are logical formulas, although these formulas are very simple. That is, these formulas are simply *atoms* and thus do not incorporate logical connectives.[7] In a similar way, the first version of 3APL [Hindriks et al., 1999b] could be categorized as a language having a logic-based representation of goals (see Chapter 3).

As another example of a logic-based approach, we mention [Simon et al., 2006], in which a *hierarchical* representation for the goals and subgoals of an agent is proposed. They propose a so-called Goal Decomposition Tree (GDT), in which the nodes represent goals and subgoals, and the

[7] In [Bordini et al., 2005a], Jason [Bordini et al., 2005b], which is an implementation of an interpreter for an extended version of AgentSpeak(L), is categorized as a logic-based language.

leaf nodes may also contain plans. The root forms an agent's top-level goal, while the other goal nodes represent subgoals that need be achieved in order to achieve the top-level goal, or other subgoals. Each goal node contains the name of the goal, and a logical formula specifying the desired situation. The aim of the authors is to use a GDT for modelling an agent, and for verifying whether the agent will achieve its top-level goal. A number of other logic-based approaches for the representation of goals will be discussed in Section 5.3.3, in which we discuss the representation of goals that may interact.

Compared with non-logic-based approaches, logic-based approaches are more likely to bridge the gap between BDI logics and agent programming frameworks. However, it remains difficult to establish an exact relation between an agent programming language and a BDI logic. One of the reasons is that the semantics of the latter is based on a Kripke-style possible worlds semantics, while the representation of beliefs and goals in agent programming languages is done differently, for practical purposes.

One way of tackling this issue, is by investigating properties of goals in agent programming frameworks, and comparing these with properties of goals in BDI logics. This can be done by introducing a logical language of goals in an agent programming language, as was done in [Hindriks et al., 2001], and, following that paper, in Chapters 2 and 4. By means of such a language, one can express, e.g., that p is a goal ($\mathbf{G}p$) or that p is not a goal ($\neg\mathbf{G}p$). In [Hindriks et al., 2001], it is shown, e.g., that this $\mathbf{G}$ operator does not obey the $\mathbf{K}$ axiom, but it does satisfy $\neg\mathbf{G}\bot$. The former is thus in contrast with BDI logics, in which goals do obey this axiom (see Section 5.1.2). We refer to Chapter 4 for a further discussion of such issues.

Besides introducing a logical language of goals in the programming language itself, one can also define a specification language including constructs for expressing that an agent has a certain goal or desire, that is not used in the programming language. When using such a language, it needs to be defined when, e.g., $\mathbf{G}p$ is true, given the agent program. The semantics of the specification language thus needs to be defined on the basis of the components of the agent program, in order to link the specification language to the agent programming language. This approach is taken in [Bordini and Moreira, 2004], in which a BDI logic for AgentSpeak(L) is defined. The properties that are investigated are mainly properties regarding the relation between the different mental attitudes. In [Hindriks et al., 2001], a temporal logic is proposed for proving properties of agents written in the cognitive agent programming language GOAL (see Section 5.4.1). This temporal logic is not used in the programming language itself, but it does build on the logical language of goals that *is* used in the programming language.

Properties as just discussed are static properties, as explained in Section 5.1.2. Dynamic properties of goals in agent programming frameworks will be treated in Section 5.4.2.

Non-Logic-Based

As an important example of a framework not using a logic-based representation of goals, we discuss Jadex. In Jadex, goals are represented using *XML* [Braubach et al., 2005]. Jadex is designed to be used "not only by AI experts, but also by the normally skilled software developer" [Pokahr et al., 2005b], which is why Jadex is based on established techniques such as XML.

Jadex supports various types of goals (see Section 5.4.2), which can be specified using the corresponding XML tags. Depending on the type of goal, it may contain various elements, such as a creation condition specifying when a new goal instance is created, a context condition that describes when a goal's execution should be suspended, and a drop condition that defines when a goal instance is removed. If a goal's creation condition becomes true, a goal object is created on the basis of the XML specification. The language for testing whether the agent has a certain belief, supports OQL (Object Query Language) constructs. OQL is a query language that is simpler than SQL, and it can be used for querying an object-oriented database. Goals can also be created by executing an appropriate statement in the plans of the agent.

Another example of a framework incorporating goals that is not based on logic, is JAM. When specifying a JAM agent, one can provide a simple *textual specification* of the agent's goals. As in Jadex, various types of goals can be specified. The specification of a goal consists of a goal's type, its name, possibly several parameters, among which a number expressing the utility of achieving the goal. Goals are used by the reasoning engine of the system to select appropriate plans. Goals can also be created through the execution of plans, comparable with the creation of subgoals in AgentSpeak(L). The behavior of subgoals however differs from that of top-level goals, i.e., the goals an agent is endowed with at start-up. This is further explained in Section 5.4.2. In [Huber, 1999], the authors do not elaborate on how the system checks whether a goal is achieved. However, given the simple representation of goals and beliefs (called "world model" in the cited paper), this check presumably does not involve logical reasoning.

5.3.3 Interacting Goals

An important issue related to the representation of goals, is the fact that goals may interact in various ways. Generally, an agent might have multiple goals. Some of these goals may be incompatible, in the sense that they interact in negative ways if being pursued simultaneously. For example, pursuing a goal p and a goal $\neg p$ at the same time is not very likely to yield satisfactory results. On the other hand, goals may interact in positive ways, in the sense that plans for different goals may have common subgoals. This can be exploited by scheduling the actions of the agent in order to take advantage of this. Research addressing these issues has resulted in new ways of representing goals. In this section, we

will discuss this research.

Hierarchies

In [Thangarajah et al., 2003b, Thangarajah et al., 2003a], Thangarajah et al. propose a *hierarchical* representation for enabling reasoning about goal interactions. Goals are represented as nodes in a so-called goal-plan tree. Besides nodes representing goals, a goal-plan tree also contains nodes representing plans. The root of the tree is a goal. The children of a goal node are plans (this is thus in contrast with the hierarchical representation of Simon et al. as discussed in Section 5.3.2), representing the alternative plans that can be used for achieving the goal. The children of a plan node are goals, representing the subgoals of the plan. The root thus forms an agent's top-level goal, while the other goal nodes represent subgoals of the plans that can be used to achieve the top-level goal, or subgoals.

Each node contains information that can be used for reasoning about goal interaction. Among other things, plan nodes contain a pre-condition which should be true before the agent can start execution of the plan, an in-condition which should be true during execution of the plan, and a set of effects representing the result of the execution of the actions in the plan. Goal nodes contain, among other things, an in-condition which should be true as long as the agent is pursuing the goal, and a condition expressing the situation the agent wants to achieve.

This information can be used for detecting and avoiding *interference* between goals [Thangarajah et al., 2003a]. Interference can occur because the execution of a plan to achieve one goal, causes the in-condition of another goal or plan to be false, or because a previously achieved effect is made false through the execution of another plan, before a plan or goal that relies on it begins executing. Using the information contained in the nodes of the goal-plan trees, these kinds of potential interferences can be detected. The execution of plans for achieving goals can then be scheduled such, that goals that have possibly interfering plans, are not pursued in parallel.

The information contained in the nodes of the goal-plan trees can also be used for detecting and exploiting *positive interactions* among goals [Thangarajah et al., 2003b]. Positive interactions occur if the plans of two goals bring about the same effect. In that case, the plans of these goals can be merged, hence reducing the overall cost of the pursuit of goals.

Default Logic

Besides the work by Thangarajah et al. in which a hierarchical representation of goals and plans is used, there is a line of research addressing the representation and handling of conflicting goals that builds on *default logic*. Our research in this direction has been discussed in Chapter 4, in which we also address work of

others along these lines. In this research, positive interactions among goals are not addressed. Also, no information about plans is used for deriving whether goals are conflicting or not.

The representation of and reasoning with logically inconsistent goals has also been addressed in [Hindriks et al., 2001] and Chapter 4 without using default logic.

Inhibition Relation

Concluding this section, we discuss an approach for representing that goals are conflicting, which has been proposed in the context of the Jadex framework [Pokahr et al., 2005a]. It is similar to approaches based on default logic, in the sense that it does not involve reasoning about the results of plan execution. Instead, it is up to the programmer to represent explicitly that a certain goal may be conflicting with another.

This is achieved by introducing so-called inhibition arcs between goals. These inhibition arcs are required to form a directed acyclic graph, in order to avoid infinite deliberation loops. Essentially, the reasoning engine makes sure that only the maximal goals with respect to this inhibition relation, i.e., those goals that are not inhibited by any other goal, are pursued. If a new goal is created that inhibits goals that the agent is currently trying to achieve, the reasoning engine makes sure that those goals are suspended. These goals can be pursued again if the inhibiting goal is dropped. The specification of the inhibition relation can be done in XML, as part of the specification of goals.

This approach differs from the approaches based on default logic (besides the fact that it is not based on logic), in that it does not allow to specify that two goals inhibit *each other*. If two goals inhibit each other, it requires the programmer to make a choice with respect to which goal should be pursued, if both are generated. In the approach of [van Riemsdijk et al., 2005b], it is possible to express that two goals should not be pursued simultaneously, without expressing that one is more important than another.

Besides the specification of an inhibition relation, the programmer can in Jadex also specify the cardinality of goals. The cardinality of a goal is a number that can be used to constrain the number of instances of this goal that can be pursued at the same time. The reasoning engine makes sure that the number of instances of a goal that the agent is trying to achieve at a certain point in time, does not exceed the goal's cardinality. This can be useful if one wants to prevent an agent from pursuing several goal instances concurrently, because the plans for achieving these goals are interfering.

5.4 Behavior

In Section 5.3, we have discussed ways in which goals have been represented in agent programming frameworks. In this section, we focus on issues with respect to the dynamic behavior of goals at run-time. We address in particular two aspects of this issue, i.e., the distinction between procedural and declarative goals (Section 5.4.1), and the dropping and adopting of goals during execution of the agent (Section 5.4.2). While the distinction between procedural and declarative goals is also closely related to, in particular, the dropping of goals, we have decided to treat the former issue in a separate section, because it has been very important in research on goals in agent programming.

5.4.1 Procedural and Declarative Goals

Roughly speaking, a *procedural goal* is the goal to execute an action or sequence of actions, and a *declarative goal* is the goal to reach a certain state of affairs. A declarative goal thus describes a desired situation. Procedural goals have also been called goals-to-do or plans, while declarative goals have been called goals-to-be [Hindriks et al., 1999b, van Riemsdijk et al., 2003b]. The importance of distinguishing actions and propositions has also been recognized early on in the philosophical literature, such as, e.g., in work on deontic logic [Castañeda, 1981]. Also, Cohen and Levesque make a distinction between the intention to do an action, and the intention to realize a certain state [Cohen and Levesque, 1990].

In the area of cognitive agent programming, it is generally accepted that an agent, being an acting entity, should at least be endowed with procedural goals. There is, however, a growing body of work in which it is argued from various perspectives that declarative goals are also important. The motivations that have been given for why the incorporation of declarative goals in agent programming frameworks is important, can roughly be divided into two categories, i.e., theoretical and practical motivations.

From a theoretical perspective, it is argued that the incorporation of a declarative perspective on goals in agent programming languages is important in order to bridge the gap between agent logics and these programming languages [Hindriks et al., 2001, Winikoff et al., 2002, van Riemsdijk et al., 2003b, Dastani et al., 2004]. Since in agent logics a goal is a declarative concept, the incorporation of declarative goals in agent languages is viewed as a necessary prerequisite for bridging this gap. It is perhaps not surprising that such research uses a logic-based representation of goals.

From a practical perspective, it is argued that declarative goals allow a decoupling of plan execution and goal achievement [Winikoff et al., 2002, van Riemsdijk et al., 2005a, Braubach et al., 2005]. The fact that declarative goals represent a state that is to be reached, can be used for deciding whether a plan was successful in achieving a certain goal, or not. If the goal has not been reached, the agent should continue to try

to reach the goal, e.g., by executing a different plan.[8] Also, declarative goals facilitate reasoning about interferences among goals (see [Winikoff et al., 2002, Thangarajah et al., 2003a] and Chapter 4). Further, in the case of a logic-based representation of goals, declarative goals allow the derivation of new goals from explicitly represented existing goals by means of logical reasoning [van Riemsdijk et al., 2003b, Dastani et al., 2006]. This can be advantageous from a modeling perspective when specifying which plan should be executed if the agent has a certain goal (consider the plan selection rule in the example of Section 2.3).

While there is thus quite some research addressing declarative goals in agent programming, a clear analysis of what exactly constitutes a declarative goal has not been carried out. There is no precise specification of when a goal may be considered a declarative goal, and when it should be termed a procedural goal. Different researchers seem to have different viewpoints with respect to this. In this section, we aim to provide a starting point for an analysis of what constitutes a declarative goal, by distinguishing various kinds of declarative goals. Generally, a goal that is not classified as a declarative goal, can be considered a procedural goal. As will become clear, most of the kinds of declarative goals are described in terms of their dynamic behavior, which is why we treat this issue in this section.

We do not aim to argue that one approach to declarative goals is always better than another. We do feel it is important that researchers recognize that there *are* different perspectives on this issue, and that it is acknowledged that one kind of declarative goal might not be equivalent with another (e.g., in terms of behavior or expressivity). In particular, we would encourage researchers, when addressing declarative goals, to be specific as to what this term means to them. We do think a thorough comparison of the various approaches against one another is important in order to establish the theoretical and practical benefits of one approach over another. However, such detailed comparisons are beyond the scope of this chapter.

Describing a Desired State

A requirement that all approaches considering declarative goals seem to agree upon, is that declarative goals *describe a state that the agent desires to reach.* In this sense, subgoals as occurring in the plans of 3APL agents (see Chapter 3) can be considered declarative goals, if they are interpreted as representing a desired state. In the same way, the construct `!p` as used in AgentSpeak(L) (see Section 5.3.1) and PRS can be considered a declarative goal, since it is generally interpreted as representing that the agent should achieve a state in which `p` is the case.

[8] This point is also related to the issue of bridging the gap, since it is concerned with commitment strategies that express when an agent may drop its goal [Winikoff et al., 2002] (see Sections 5.1.2 and 5.4.2).

Behaving as a Declarative Goal

While specifying a declarative goal as a goal to reach a certain state gives some indication of what a declarative goal should be, it does not say anything about the *behavior* of such a goal during execution of the agent. That is, it does not say whether the fact that the goal represents a state, is *used* somehow for specifying the behavior of the agent. Given that we want to be more specific about how a declarative goal should behave, the question is, of course, what kind of behavior this should be.

In most research on declarative goals, the fact that the goal represents a state is used for deciding when to drop the goal. That is, a goal may not be dropped until it is achieved, i.e., until the state expressed by the goal is reached.[9] This is, for example, the approach taken in [Hindriks et al., 2001], and, following that paper, in Chapter 2.

From this perspective, the subgoals of the language of Sections 3.1 and 3.2 can also be viewed as declarative goals, since they can only be removed from the plan if they are believed to be achieved. Moreover, the subgoals of 3APL can also be viewed as declarative goals in this sense, but *only* if the plans of the 3APL agent are programmed such that they behave as the subgoals of Sections 3.1 and 3.2. We can thus not say in general that subgoals of 3APL behave as declarative goals, but it varies from agent to agent. In the context of Jason[10], similar research has been done [Hübner et al., 2006]. In that paper, several transformations on the plans of Jason agents are proposed, such that the subgoals of the resulting agent behave as declarative goals.[11] It is, however, not shown formally that the resulting agent behaves according to a certain specification.

The goals of [Winikoff et al., 2002] also behave as declarative goals, although the conditions for dropping the goal do not only depend on whether the desired state as represented by the goal is achieved. We refer to Section 5.4.2 for a further explanation of this. The goals of the Jadex framework [Braubach et al., 2005] can also be viewed as behaving as declarative goals, although the desired state is not represented using logical formulas, which is in contrast with most other approaches addressing declarative goals. Nevertheless, it can be checked whether the desired state is believed to be reached. Further

[9] Here, we focus on the behavior of achievement goals, i.e., goals that are dropped once achieved. One can, however, consider other kinds of behavior in which the fact that a goal represents a state is used. For example, in Section 5.4.2 we also consider the behavior of so-called maintenance goals. These can be analyzed in order to identify when they could be called declarative goals in a similar way as we do for achievement goals in this section.

[10] Recall that Jason is an implementation of an interpreter for an extended version of AgentSpeak(L)

[11] Each transformation yields a different variant of declarative behavior of goals. One such transformation, e.g., results in plans failing if a subgoal is not reached after executing a plan for achieving it. This transformation does not yield an agent that only drops its subgoals if they are believed to be achieved. Nevertheless, the fact that the subgoal represents a desired state *is* used.

examples of research in which goals behave as declarative goals can be found in [van Riemsdijk et al., 2003b, Dastani et al., 2004, Dastani et al., 2006].

Having a Declarative Goal Semantics

Above, we have considered a definition of declarative goals that is based on whether the fact that they represent a desired state, is reflected in their behavior. While this can be viewed as a sufficient characterization of declarative goals, some might argue for a still stronger definition. That is, one could argue that the *declarative behavior of goals should be enforced* by the semantics of the programming language or by the framework. Viewed from this perspective, the subgoals of 3APL and Jason (and AgentSpeak(L)) cannot be classified as declarative goals. The other discussed approaches can still be viewed as incorporating declarative goals under this definition.

Using Logical Reasoning

We conclude this section with an even stronger definition of declarative goals, which one might want to adopt especially in the context of bridging the gap. This definition considers only approaches that use logic for representing goals, and requires a more sophisticated use of the fact that goals are represented as logical formulas.

In particular, the representation of goals should be based on a full-fledged logical language, such as propositional logic. The subgoals of the language of Sections 3.1 and 3.2 and of 3APL and Jason thus would not qualify as declarative goals in this sense, since these subgoals are simply atoms. Further, the programming framework should use *logical reasoning* for establishing the goals of the agent. Subgoals as used in the plans of agents do not fulfill this requirement, since these goals are generally not considered in relation to other goals, and are not used as input for a logical reasoning process. Goals as used in [Winikoff et al., 2002] are represented using propositional logic. However, they are not used for logical reasoning, although the paper contains some suggestions on how this could be done.

Approaches that *are* based on logical reasoning with goals are mainly approaches inspired directly or indirectly by [Hindriks et al., 2001], such as [van Riemsdijk et al., 2003b, Dastani et al., 2004, van Riemsdijk et al., 2005a, Dastani et al., 2006]. The semantics of Chapter 4 and other approaches for representing goals that are based on default logic also rely on logical reasoning for establishing the goals of an agent [Thomason, 2000, Broersen et al., 2002, Dastani and van der Torre, 2004]. The focus of these approaches is not so much on the dynamic behavior of goals, but the proposed frameworks do make use of the fact that goals are represented using logical formulas. The approach of [Sardina and Shapiro, 2003] is another example of an approach using logical reasoning in the context of goals.

If agent programming languages incorporate a representation and semantics of declarative goals that is strongly based on logic, it is more likely that a relation between the programming language and BDI logics can be established. The added expressivity provided by such logic-based representations and semantics can, however, also be beneficial from a practical perspective.

5.4.2 Dropping and Adopting

As noted in Section 5.1.2, an important aspect of goals is their behavior over time, i.e., when are goals *adopted* and when are they *dropped* again. In BDI logics, the focus is on the latter. This is perhaps not surprising, since these logics aim to formalize BDI philosophy, and an important aspect of BDI philosophy is that agents should be *committed* towards their intentions. That is, agents should stick to their intentions, unless there is a good reason for giving them up.

While this issue of commitment has been investigated in the context of *intentions* in BDI logics, it has been taken up by developers of agent programming frameworks in the context of *goals*. In particular, the commitment strategies as proposed in [Rao and Georgeff, 1991] for intentions, have been an inspiration for implementing the behavior of goals in agent programming frameworks.

Dropping

These commitment strategies for agent programming frameworks are concerned with conditions under which a goal may be dropped, such as if it is believed to be achieved. Goals that may be dropped if believed to be achieved, are generally called *achievement goals* in agent programming frameworks.[12] In logic-based approaches working with a goal base, achievement goals are implemented by removing a goal from the goal base, once the agent believes the goal to be achieved. In JAM, achievement goals are implemented in a similar way. With respect to JAM, we mention that the framework makes a distinction with respect to the level of commitment an agent has towards its top-level goals, i.e., the goals an agent is endowed with at start-up, and the subgoals as generated from the plans of the agent. The agent's top-level achievement goals are maintained until believed to be achieved, while the subgoals are dropped if a plan to achieve a subgoal fails.

These implementations of achievement goals are implementations of *blind commitment*, i.e., goals are maintained until the agent believes it has achieved them. Implementing blindly committed agents is thus relatively straightforward.

[12] The subgoals of the form `!p` as used in the plans of AgentSpeak(L) agents are also called achievement goals. Moreover, the abstract plans of 3APL [Dastani et al., 2004] and as used in the language of Chapter 2 were also called achievement goals in the first version of 3APL [Hindriks et al., 1999b]. In this section, we will however not use the term "achievement goal" to refer to a language construct of AgentSpeak(L) or 3APL.

Implementing a *single minded* agent, i.e., an agent that keeps a goal until it believes it has achieved it or believes it is impossible to achieve it, is more complicated. In order to implement this commitment strategy, an agent will, in principle, have to reason about its future. There is, however, another way to implement this commitment strategy, which does not require the agent to reason about its future.

This approach was first proposed in [Winikoff et al., 2002], and is also followed in the Jadex framework and in [Dastani et al., 2006]. The idea is, to endow goals with a so-called *failure condition*, which represents a situation in which it will not be possible for the agent to achieve its goal. If this failure condition becomes true, the agent should drop the goal. This is thus a condition that should be specified by the agent programmer. That is, it is up to the agent programmer to think of situations in which the agent will not be able to reach a goal, rather than having the agent reason about this. Open minded agents are generally not considered in agent programming frameworks.

Perhaps following BDI logics, achievement goals have traditionally been the focus of research regarding goals in agent programming frameworks. There is however another type of goal that has been identified, which is referred to as *maintenance goal*. A maintenance goal describes a situation that the agent should maintain, i.e., the agent should make sure that this situation continues to hold. Maintenance goals should thus not be dropped. In [Cohen and Levesque, 1990], it is mentioned that this distinction exists, but state that they will not be concerned with maintenance goals.

Agent programming frameworks that have incorporated maintenance goals are PRS [Ingrand et al., 1992], JAM, Jadex, and [Dastani et al., 2006]. In PRS, a language construct is proposed for maintenance goals, that is similar to the construct for achievement goals as used in PRS and AgentSpeak(L). However, very little is said in the cited paper about the meaning of such a language construct. In JAM, the agent starts to pursue maintenance goals whenever the desired situation as specified by the maintenance goal becomes unsatisfied. This is thus comparable with the behavior of achievement goals, except that achievement goals are removed once they are achieved, while maintenance goals are not dropped.

The implementation of maintenance goals in Jadex is somewhat more involved. As in JAM, maintenance goals contain a condition, i.e., the so-called maintain condition, which triggers the execution of plans if it becomes false. The agent can stop executing plans once the maintain condition is reached. Besides this maintain condition, maintenance goals may also contain a target condition. If such a target condition is specified, it is the target condition, rather than the maintenance condition, that specifies when the agent may stop executing plans for the maintenance goal, i.e., the agent can stop the execution of plans for the maintenance goal if this target condition is reached. In this case, the execution of plans is thus started once the maintain condition becomes false, and is stopped once the target condition becomes true. This mechanism can,

e.g., be used for an agent that should refuel once the level of fuel in its tank drops below 20 %. Once it starts refueling, it should continue until its tank is completely full again.[13] The maintain condition can then be used to make sure the agent starts refueling, while the target condition specifies when the agent may stop.

In [Dastani et al., 2006], maintenance goals have a triggering condition and a maintain condition. The triggering condition indicates when the agent should take action in order to ensure that the maintain condition continues to hold. The maintain condition is used to select appropriate plans. The triggering condition of Dastani et al. is thus comparable with the maintain condition of Jadex, and the maintain condition of Dastani et al. is comparable with the target condition of Jadex (although there are some subtle differences).

Besides achievement goals and maintenance goals, another type of goal has been introduced in JAM, Jadex and [Dastani et al., 2006], i.e., a so-called *perform goal*. A perform goal can be viewed as a procedural goal, since it is not related to a world state that should be achieved,[14] but only to actions that should be executed. The dynamic behavior of a perform goal is simple, since it is dropped once it is executed.[15]

Finally, we mention that GOAL [Hindriks et al., 2001] has an action `drop`(ϕ) for dropping a goal ϕ. If such an action is executed, all goals from which ϕ logically follows are removed from the goal base.

Adopting

Implementations of the behavior of goals with respect to when they are dropped, are mainly based on the idea of commitment strategies in BDI logics. On the other hand, mechanisms for *adopting* goals have been investigated primarily in the context of agent programming frameworks.
In [van Riemsdijk et al., 2005a], an overview is presented of motivations for goal adoption that have been suggested in the literature. The cited paper distinguishes between *internal* and *external* motivations for goal adoption.

As internal motivations, the generation of concrete goals from built-in abstract goals as suggested in [Dignum and Conte, 1997] is mentioned. The idea is, that abstract goals are not really achievable, but they can be approximated through the achievement of concrete goals. In [van Riemsdijk et al., 2005a], an implementation of this kind of goal adoption is proposed. Another internal motivation for adopting goals are desires, as used, e.g., in the BOID framework. Desires in BOID are conditional on the agent's beliefs or goals.

The beliefs and goals of an agent can, in general, be seen as internal reasons

[13] Similar examples were used in [Braubach et al., 2005] and [Dastani et al., 2006].

[14] In [Dastani et al., 2006], perform goals *are* related to a world state, but this is used only for selecting appropriate plans.

[15] In Jadex and [Dastani et al., 2006], perform goals may also be endowed with a flag indicating that the perform goal should be executed continuously.

for adopting (new) goals. In Jadex, this approach is followed in that goals can be endowed with a so-called creation condition. This can be a condition on beliefs (or goals), and if it becomes true, an instance of the goal as specified in the XML file is generated. A goal can also be adopted in Jadex by executing a specific action in the plan of an agent. GOAL also has an action `adopt`(ϕ) for adopting a goal ϕ. If such an action is executed, ϕ is added to the goal base.[16]

In [van Riemsdijk et al., 2005b], goals can be conditional on beliefs and other goals, as in Jadex. However, the way in which these conditional goals are used is different from the approach taken in Jadex. That is, in the former, these conditional goals are used for defining the semantics of a logical language of goals, based on the framework of default logic. In the latter, conditional goals are used for the creation of goal instances at run-time. These goal instances are placed in the goal base, once they are created.

As external motivations for the adoption of goals, [van Riemsdijk et al., 2005a] addresses norms and obligations, and requests from other agents. In the BOID framework, obligations can be used for the generation of goals, in a similar way in which desires can be used. Like desires, obligations are conditional on beliefs or goals. An implementation of the adoption of a goal because of a request from another agent, has been proposed in the context of AgentSpeak(L) in [Moreira et al., 2004]. In that paper, a formal semantics of various communication acts for AgentSpeak(L) agents is presented. These communication acts include requests for achieving a goal. If one agent sends such a request to another agent, the goal that the receiving agent is requested to achieve, is placed in its event base. That is, if the requesting agent has power over the receiving agent.

5.5 Conclusion

In this chapter, we have presented an overview of various approaches that in some form address the incorporation of goals in agent programming frameworks. We have started out by considering goals and related motivational attitudes in BDI philosophy and BDI logics, which lie at the basis of research into cognitive agent programming. This was followed by an argumentation of why we think goals are important in agent programming. Then, we have identified and discussed important aspects with respect to which approaches for representing goals in agent programming frameworks differ. Finally, we have discussed the dynamic behavior of goals in agent programming frameworks. In particular, we have provided an analysis of various views on what constitutes a declarative goal.

In this chapter, we have discussed many different approaches to incorporating goals in agent programming frameworks. In our view, there is not one "right" way of doing this. It will depend on the reason for wanting to

[16] That is, if ϕ is consistent and not believed to be achieved.

use goals and on the context, which is the most appropriate approach (cf. [Rao and Georgeff, 1998], in which formalizations of various different BDI systems are presented).

Nevertheless, we believe it is important to investigate exactly how the various approaches are related to one another. Does one approach differ from another, e.g., in terms of expressiveness, or are they essentially the same or is one subsumed by another? We hope that this chapter can be a starting point for such an analysis, which will clarify how various approaches to goals differ from one another and what their merits are. This will provide a better understanding of the essence of different kinds of goals, which will eventually enable agent programmers to choose the approach most suitable for their application.

Acknowledgements

We would like to thank Koen Hindriks for his valuable comments on an earlier version of this chapter.

PART II

PLAN REVISION

Chapter 6

Semantics of Plan Revision

This chapter is based on [van Riemsdijk et al., 2003a, van Riemsdijk et al., 2004, van Riemsdijk et al., 2006c]. Plan revision rules have been introduced in Chapter 2 (Definition 2.4). As explained in Section 2.3.2, plan revision rules can provide the agent with added flexibility, since they allow the agent to revise its plan if the circumstances demand this. The introduction of these plan revision capabilities now gives rise to interesting issues concerning the *semantics of plan execution*, the exploration of which is the topic of this chapter.

An investigation of these semantic issues is important in order to get a better understanding of the language. Furthermore, certain semantic properties are especially relevant when it comes to verification. In the context of plan revision rules, the important semantic property to consider is *compositionality*. Generally, when proving that a program satisfies a specification, this is done by proving properties of the parts of which the program is composed. This can be done if the semantics of programs can be defined compositionally, i.e., by defining the semantics of a composed program in terms of the semantics of the parts of which the program is composed. In this chapter, we investigate this issue of compositionality in the context of the semantics of the execution of plans, where plans can be revised during execution by means of plan revision rules.

The cognitive agent programming language we investigate in this chapter is the language of Chapter 2, without goals and plan selection rules and with some other minor modifications. The reason for leaving out goals is that in this chapter we focus on the semantics of plan execution, for the treatment of which only beliefs and plans will suffice. The language under consideration is essentially a propositional and otherwise slightly simplified version of the first version of 3APL [Hindriks et al., 1999b], and we will in the rest of this chapter refer to our language simply as "3APL". The language will be specified in detail in Section 6.1.1.

The approach we take to investigating the semantics of plan execution in

3APL, is to introduce a *meta-language* on top of 3APL. Such a meta-language has constructs for applying a plan revision rule and executing an action. The data on which the meta-program operates are a plan and a belief base. Such a meta-program can be viewed as an *interpreter* for 3APL.

We provide a semantics for this meta-language using a transition system (Section 6.2). Such a semantics is called an *operational semantics*. We link this meta-language to object-level 3APL by showing that the meta-level operational semantics of a certain meta-program is equivalent with the object-level operational semantics of 3APL (Section 6.3). As it turns out, providing a compositional semantics for object-level 3APL is problematic. Instead, we provide a compositional or *denotational semantics* for the meta-language (Section 6.4). We prove that this meta-level denotational semantics is equivalent with the meta-level operational semantics (Section 6.5). In Section 6.5.2, we discuss whether the specification of a denotational semantics for the meta-language can be used for specifying a denotational semantics for object-level 3APL.

For regular procedural programming languages, studying a specific meta-language or interpreter language is in general not very interesting. In the context of agent programming languages it however *is*, for several reasons. First of all, 3APL and agent programming languages in general are non-deterministic by nature. In the case of 3APL for example, it will often occur that several plan revision rules are applicable at the same time. Choosing a rule for application (or choosing whether to execute an action from the plan or to apply a rule if both are possible) is the task of a 3APL interpreter. The choices made affect the outcome of the execution of the agent. In the context of agents, it is interesting to study various interpreters, as different interpreters will give rise to different *agent types*. An interpreter that for example always executes a rule if possible, thereby deferring action execution, will yield a thoughtful and passive agent. In a similar way, very bold agents can be constructed or agents with characteristics anywhere on this spectrum. These conceptual ideas about various agent types fit well within the agent metaphor and therefore it is worthwhile to study an interpreter language and the interpreters that can be programmed in it (see also [Dastani et al., 2003]).

Further, as pointed out by Hindriks et al. [Hindriks et al., 1999a], differences between various agent languages often mainly come down to differences in their meta-level reasoning cycle or interpreter. To provide for a *comparison* between languages, it is thus important to separate the semantic specification of object-level and meta-level execution.

6.1 Syntax

6.1.1 Object-Level

In this section, we present the propositional version of 3APL that we use in this chapter. Belief bases and plans are as in Chapter 2 (Definitions 2.1 and 2.3). Note that we use ψ for formulas from $\mathcal{L}$, rather than ϕ. The symbol ϕ will be used in a different context in this chapter.

Definition 6.1 *(belief base)* Assume a propositional language $\mathcal{L}$ with typical formula ψ and the connectives $\wedge$ and $\neg$ with the usual meaning. Then the set of belief bases Σ with typical element σ is defined to be $\wp(\mathcal{L})$.

Definition 6.2 *(plan)* Assume that a set $\mathsf{BasicAction}$ with typical element a is given, together with a set $\mathsf{AbstractPlan}$ with typical element p. Let $c \in (\mathsf{BasicAction} \cup \mathsf{AbstractPlan})$. Then the set of plans Plan with typical element π is defined as follows.

$$\pi ::= a \mid p \mid c; \pi$$

An empty plan will in this chapter be denoted by E. In contrast with 3APL as presented in [Hindriks et al., 1999b], we exclude non-deterministic choice and test from plans for reasons of presentation and technical convenience. This is no fundamental restriction as non-determinism is introduced by plan revision rules. Furthermore, tests can be modeled as basic actions that do not affect the belief base if executed.

Plan revision rules are as in Chapter 2 (Definition 2.4), except that the belief condition is a formula from $\mathcal{L}$, rather than a belief formula with $\mathbf{B}$ operators. This is for reasons of simplicity, and the choice of language for the belief condition does not influence the issue of compositionality that we investigate in this chapter. Note that plan revision rules are typically denoted by ρ. Also, it is important to note that π_h may not be the empty plan, since the empty plan is not included in the definition of Plan.

Definition 6.3 *(plan revision rules)* A plan revision rule ρ is a triple $\pi_h \mid \psi \rightsquigarrow \pi_b$ such that $\psi \in \mathcal{L}$, $\pi_h, \pi_b \in \mathsf{Plan}$.

The definition of a 3APL agent is presented below. It is important to note that a 3APL agent has an initial plan, which is in contrast with the agent program of Chapter 2 (Definition 2.5). The reason for this is that 3APL agents do not have goals on the basis of which the agent can select plans, and so the agent is endowed with a plan at start-up.

Definition 6.4 *(3APL agent)* A 3APL agent $\mathcal{A}$ is a tuple $\langle \pi_0, \sigma_0, \mathsf{Rule}, \mathcal{T} \rangle$, where Rule is a finite set plan revision rules and $\mathcal{T} : (\mathsf{BasicAction} \times \Sigma) \rightarrow \Sigma$ is a partial function, defining belief update through action execution.

In the following, when referring to agent $\mathcal{A}$, we will assume this agent to have a set of plan revision rules Rule and a belief update function $\mathcal{T}$.

In this chapter, configurations consist of a plan and a belief base, and will be called *mental states*.

Definition 6.5 *(mental state)* Let Σ be the set of belief bases and let Plan be the set of plans. Then $\mathsf{Plan} \times \Sigma$ is the set S with typical element s of possible mental states of a 3APL agent. A mental state with plan π and belief base σ will be denoted as $\langle \pi, \sigma \rangle$. If $\langle \pi_0, \sigma_0, \mathsf{Rule}, \mathcal{T} \rangle$ is an agent, then $\langle \pi_0, \sigma_0 \rangle$ is the initial mental state of the agent.

6.1.2 Meta-Level

In this section, we define the meta-language that can be used to write 3APL interpreters. The programs that can be written in this language will be called *meta-programs*. Like regular imperative programs, these programs are state transformers. The kind of states they transform however do not simply consist of an assignment of values to variables like in regular imperative programming, but the states that are transformed are 3APL mental states. In Section 6.2.2, we will define the transition system with which we will define the operational semantics of our meta-programs. We will do this using the concept of a *meta-configuration*. A meta-configuration consists of a meta-program and a mental state, i.e., the meta-program is the procedural part and the mental state is the "data" on which the meta-program operates.

The basic elements of meta-programs are the *execute* action and the $apply(\rho)$ action (called *meta-actions*). The *execute* action is used to specify that a basic action from the plan of an agent should be executed. The $apply(\rho)$ action is used to specify that a plan revision rule ρ should be applied to the plan. Composite meta-programs can be constructed in a standard way.

Below, the meta-programs and meta-configurations for agent $\mathcal{A}$ are defined. An empty meta-program will be denoted by E.[1]

Definition 6.6 *(meta-programs)* We assume a set $Bexp$ of boolean expressions with typical element b. Let $b \in Bexp$ and $\rho \in \mathsf{Rule}$, then the set $Prog$ of meta-programs with typical element P is defined as follows:

$$P ::= execute \mid apply(\rho) \mid \texttt{while}\ b\ \texttt{do}\ P\ \texttt{od} \mid P_1; P_2 \mid P_1 + P_2.$$

Definition 6.7 *(meta-configurations)* Let $Prog$ be the set of meta-programs and let S be the set of mental states. Then $Prog \times S$ is the set of possible meta-configurations. A meta-configuration with meta-program P and mental state s will be denoted as $\langle P, s \rangle$.

[1] Note that an empty plan, as well as an empty meta-program, are denoted by E.

6.2 Operational Semantics

In this section, we present the operational semantics of 3APL and of the meta-language. This is done using transition systems.

6.2.1 Object-Level Transition System

The transition system for 3APL resembles the transition system of Chapter 2, where aspects related to goals are omitted. We will call this transition system the *object-level transition system*. This transition system can also be viewed as an adaptation of the transition system presented in [Hindriks et al., 1999b], to fit our simplified language.

The transition systems defined in this and the following section assume 3APL agent $\mathcal{A}$. The object-level transition system (Trans_o) is defined by the rules given below. The transitions are labeled to denote the kind of transition.

Definition 6.8 *(action execution)* Let $a \in \mathsf{BasicAction}$.

$$\frac{\mathcal{T}(a,\sigma) = \sigma'}{\langle a;\pi,\sigma\rangle \rightarrow_{execute} \langle \pi,\sigma'\rangle}$$

In the next definition, we use the operator $\bullet$ for concatenating two plans, as introduced in Chapter 2 (Definition 2.8).

Definition 6.9 *(rule application)* Let $\rho : \pi_h \mid \psi \rightsquigarrow \pi_b \in \mathsf{Rule}$.

$$\frac{\sigma \models \psi}{\langle \pi_h \bullet \pi,\sigma\rangle \rightarrow_{apply(\rho)} \langle \pi_b \bullet \pi,\sigma\rangle}$$

6.2.2 Meta-Level Transition System

The *meta-level transition system* (Trans_m) is defined by the rules below, specifying which transitions from one meta-configuration to another are possible. As for the object-level transition system, the transitions are labeled to denote the kind of transition.

An *execute* meta-action is used to execute a basic action. It can thus only be executed in a mental state, if the first element of the plan in that mental state is a basic action. As in the object-level transition system, the basic action a must be executable and the result of executing a on belief base σ is defined using the function $\mathcal{T}$. After executing the meta-action *execute*, the meta-program is empty and the basic action is gone from the plan. Furthermore, the belief base is changed as defined through $\mathcal{T}$.

Definition 6.10 *(action execution)* Let $a \in \mathsf{BasicAction}$.

$$\frac{\mathcal{T}(a,\sigma) = \sigma'}{\langle execute,(a;\pi,\sigma)\rangle \rightarrow_{execute} \langle E,(\pi,\sigma')\rangle}$$

A meta-action $apply(\rho)$ is used to specify that plan revision rule ρ should be applied. It can be executed in a mental state if ρ is applicable in that mental state. The execution of the meta-action in a mental state results in the plan of that mental state being changed as specified by the rule.

Definition 6.11 *(rule application)* Let $\rho : \pi_h \mid \psi \rightsquigarrow \pi_b \in \mathsf{Rule}$.

$$\frac{\sigma \models \psi}{\langle apply(\rho), (\pi_h \bullet \pi, \sigma)\rangle \rightarrow_{apply(\rho)} \langle E, (\pi_b \bullet \pi, \sigma)\rangle}$$

In order to define the transition rule for the `while` construct, we first need to specify the semantics of boolean expressions $Bexp$.

Definition 6.12 *(semantics of boolean expressions)* We assume a function $\mathcal{W} : Bexp \rightarrow (S \rightarrow W)$ yielding the semantics of boolean expressions, where W is the set of truth values $\{tt, ff\}$ with typical formula β.

The transition for the `while` construct is then defined in a standard way below. The transition is labeled with *idle*, to denote that this is a transition that does not have a counterpart in the object-level transition system.

Definition 6.13 *(while)*

$$\frac{\mathcal{W}(b)(s)}{\langle \texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}, s\rangle \rightarrow_{idle} \langle P; \texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}, s\rangle}$$

$$\frac{\neg\mathcal{W}(b)(s)}{\langle \texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}, s\rangle \rightarrow_{idle} \langle E, s\rangle}$$

The transitions for sequential composition and non-deterministic choice are defined as follows in a standard way. The variable x is used to pass on the type of transition through the derivation.

Definition 6.14 *(sequential composition)* Let $x \in \{execute, apply(\rho), idle \mid \rho \in \mathsf{Rule}\}$.

$$\frac{\langle P_1, s\rangle \rightarrow_x \langle P_1', s'\rangle}{\langle P_1; P_2, s\rangle \rightarrow_x \langle P_1'; P_2, s'\rangle}$$

Definition 6.15 *(non-deterministic choice)* Let $x \in \{execute, apply(\rho), idle \mid \rho \in \mathsf{Rule}\}$.

$$\frac{\langle P_1, s\rangle \rightarrow_x \langle P_1', s'\rangle}{\langle P_1 + P_2, s\rangle \rightarrow_x \langle P_1', s'\rangle} \qquad \frac{\langle P_2, s\rangle \rightarrow_x \langle P_2', s'\rangle}{\langle P_1 + P_2, s\rangle \rightarrow_x \langle P_2', s'\rangle}$$

6.2.3 Operational Semantics

Using the transition systems defined in the previous section, transitions can be derived for 3APL and for the meta-programs. Individual transitions can be put in sequel, yielding so called *computation sequences*. In the following definitions, we define computation sequences and we specify the functions yielding these sequences, for the object- and meta-level transition systems. We also define the function κ, yielding the last element of a computation sequence if this sequence is finite, and the special state $\bot$ otherwise. These functions will be used to define the operational semantics.

Definition 6.16 *(computation sequences)* The sets S^+ and S^∞ of respectively finite and infinite computation sequences are defined as follows:

$$\begin{array}{lcll} S^+ & = & \{s_1, \ldots, s_i, \ldots, s_n & \mid \quad s_i \in S, 1 \leq i \leq n, n \in \mathbb{N}\}, \\ S^\infty & = & \{s_1, \ldots, s_i, \ldots & \mid \quad s_i \in S, i \in \mathbb{N}\}. \end{array}$$

Let $S_\bot = S \cup \{\bot\}$ and $\delta \in S^+ \cup S^\infty$. The function $\kappa : (S^+ \cup S^\infty) \rightarrow S_\bot$ is defined by:

$$\kappa(\delta) = \begin{cases} \text{last element of } \delta & \text{if } \delta \in S^+, \\ \bot & \text{otherwise.} \end{cases}$$

The function κ is extended to handle sets of computation sequences as follows: $\kappa(\{\delta_i \mid i \in I\}) = \{\kappa(\delta_i) \mid i \in I\}$.

Definition 6.17 *(functions for calculating computation sequences)* The functions $\mathcal{C}_o$ and $\mathcal{C}_m$ are respectively of type $S \rightarrow \wp(S^+ \cup S^\infty)$ and $Prog \rightarrow (S \rightarrow \wp(S^+ \cup S^\infty))$.

$$\begin{array}{lcll} \mathcal{C}_o(s) & = & \{s_1, \ldots, s_n \in \wp(S^+) & \mid \quad s \rightarrow_{t_1} s_1 \rightarrow_{t_2} \ldots \rightarrow_{t_n} \langle E, \sigma_n \rangle \\ & & \quad \text{is a finite sequence of transitions in } \mathsf{Trans}_o\} \cup \\ & & \{s_1, \ldots, s_i, \ldots \in \wp(S^\infty) & \mid \quad s \rightarrow_{t_1} s_1 \rightarrow_{t_2} \ldots \rightarrow_{t_i} s_i \rightarrow_{t_{i+1}} \ldots \\ & & \quad \text{is an infinite sequence of transitions in } \mathsf{Trans}_o\} \\ \mathcal{C}_m(P)(s) & = & \{s_1, \ldots, s_n \in \wp(S^+) & \mid \quad \langle P, s \rangle \rightarrow_{x_1} \langle P_1, s_1 \rangle \rightarrow_{x_2} \ldots \\ & & & \quad \rightarrow_{x_n} \langle E, s_n \rangle \\ & & \quad \text{is a finite sequence of transitions in } \mathsf{Trans}_m\} \cup \\ & & \{s_1, \ldots, s_i, \ldots \in \wp(S^\infty) & \mid \quad \langle P, s \rangle \rightarrow_{x_1} \langle P_1, s_1 \rangle \rightarrow_{x_2} \ldots \\ & & & \quad \rightarrow_{x_i} \langle P_i, s_i \rangle \rightarrow_{x_{i+1}} \ldots \\ & & \quad \text{is an infinite sequence of transitions in } \mathsf{Trans}_m\} \end{array}$$

Note that both $\mathcal{C}_o$ and $\mathcal{C}_m$ return sequences of mental states. $\mathcal{C}_o$ just returns the mental states comprising the sequences of transitions derived in Trans_o, whereas $\mathcal{C}_m$ removes the meta-program component of the meta-configurations of the transition sequences derived in Trans_m. The reason for defining these functions in this way is that we want to prove equivalence of the object- and meta-level transition systems: both yield the same transition sequences with

respect to the mental states (or that is for a certain meta-program, see Section 6.3). Also note that for $\mathcal{C}_o$ as well as for $\mathcal{C}_m$, we only take into account infinite sequences and successfully terminating sequences, i.e., those sequences ending in a mental state or meta-configuration with an empty plan or meta-program respectively.

The operational semantics of object- and meta-level programs are functions $\mathcal{O}_o$ and $\mathcal{O}_m$, yielding, for each mental state s and possibly meta-program P, a set of mental states corresponding to the final states reachable through executing the plan of s or executing the meta-program P respectively. If there is an infinite execution path, the set of mental states will contain the element $\perp$.

Definition 6.18 *(operational semantics)* Let $s \in S$. The functions $\mathcal{O}_o$ and $\mathcal{O}_m$ are respectively of type $S_\perp \to \wp(S_\perp)$ and $Prog \to (S_\perp \to \wp(S_\perp))$.

$$\begin{array}{lcl} \mathcal{O}_o(s) & = & \kappa(\mathcal{C}_o(s)) \\ \mathcal{O}_m(P)(s) & = & \kappa(\mathcal{C}_m(P)(s)) \\ \mathcal{O}_o(\perp) & = & \mathcal{O}_m(P)(\perp) \quad = \quad \{\perp\} \end{array}$$

Note that the operational semantic functions can take any state $s \in S_\perp$, including $\perp$, as input. This will turn out to be necessary for giving the equivalence result of Section 6.5.

6.3 Equivalence of Object- and Meta-Level Operational Semantics

In the previous section, we have defined the operational semantics for 3APL and for meta-programs. Using the meta-language, one can write various 3APL interpreters. Here we will consider an interpreter of which the operational semantics will prove to be equivalent to the object-level operational semantics of 3APL. This interpreter for agent $\mathcal{A}$ is defined by the following meta-program.

Definition 6.19 *(interpreter)* Let $\bigcup_{i=1}^{n} \rho_i = \mathsf{Rule}$, let $s \in S$ and let $notEmptyPlan \in Bexp$ be a boolean expression such that $\mathcal{W}(notEmptyPlan)(s) = tt$ if the plan component of s is not equal to E and $\mathcal{W}(notEmptyPlan)(s) = ff$ otherwise. Then the interpreter can be defined as follows.

$$\texttt{while}\ notEmptyPlan\ \texttt{do}\ (execute\ +\ apply(\rho_1)\ +\ \ldots\ +\ apply(\rho_n))\ \texttt{od}$$

In the sequel, we will use the keyword interpreter to abbreviate this meta-program.

This interpreter thus iterates the execution of a non-deterministic choice between all basic meta-actions, until the plan component of the mental state is empty. Intuitively, if there is a possibility for the interpreter to execute some meta-action in mental state s, resulting in a changed state s', it is also possible to

go from s to s' in an object-level execution through a corresponding object-level transition. At each iteration, an executable meta-action is non-deterministically chosen for execution. The interpreter thus, as it were, non-deterministically chooses a path through the object-level transition tree. The possible transitions defined by this interpreter correspond to the possible transitions in the object-level transition system and therefore the object-level operational semantics is equivalent to the meta-level operational semantics of this meta-program. In the sequel we will provide some lemma's and a corollary from which this equivalence result will prove to follow immediately.

Remark 6.1 *(reactive plan revision rules)* Before moving on to proving the equivalence theorem, we have to make the following remark. The equivalence between object-level 3APL and the interpreter defined above would not hold in general if the empty plan would be part of the language of plans Plan. This would result in the possibility of Rule containing reactive rules of the form $E \mid \psi \rightsquigarrow \pi_b$. Such rules with an empty plan as the head would be applicable, regardless of the plan of the agent in a mental state, since any plan has the empty plan as a prefix (the plan π_h in Definition 6.9 would then be E).

The empty plan was chosen as a termination condition for the interpreter and the interpreter will thus stop applying rules in a mental state once the plan in this state is empty. Rule application is however still a possible transition in the object-level transition system in case of the presence of reactive rules, as these are applicable to empty plans. Having an empty plan or program as a termination condition is in line with the notion of successful termination in procedural programming languages.

We prove the equivalence result by proving a weak bisimulation between Trans_o and $\mathsf{Trans}_m(\mathsf{interpreter})$, which are defined assuming agent $\mathcal{A}$ (see the text below Definition 6.4 and Section 6.2.1). From this, we can then prove that $\mathcal{O}_o$ and $\mathcal{O}_m(\mathsf{interpreter})$ are equivalent. In order to do this, we first state the following proposition. It follows immediately from the transition systems.

Proposition 6.1 *(object-level versus meta-level transitions)*

$$\begin{array}{rlll} s & \rightarrow_{execute} & s' & \text{is a transition in } \mathsf{Trans}_o \quad \Leftrightarrow \\ \langle execute, s\rangle & \rightarrow_{execute} & \langle E, s'\rangle & \text{is a transition in } \mathsf{Trans}_m \end{array}$$

$$\begin{array}{rlll} s & \rightarrow_{apply(\rho)} & s' & \text{is a transition in } \mathsf{Trans}_o \quad \Leftrightarrow \\ \langle apply(\rho), s\rangle & \rightarrow_{apply(\rho)} & \langle E, s'\rangle & \text{is a transition in } \mathsf{Trans}_m \end{array}$$

A weak bisimulation between two transition systems in general, is a relation between the systems such that the following holds: if a transition step can be derived in system one, it should be possible to derive a "similar" (sequence of) transition(s) in system two and if a transition step can be derived in system two, it should be possible to derive a "similar" (sequence of) transition(s) in system

one. To explain what we mean by "similar" transitions, we need the notion of an idle transition. In a transition system, certain kinds of transitions can be labelled as an idle transition, for example transitions derived using the while rule (Definition 6.13). These transitions can be considered "implementation details" of a certain transition system and we do not want to take these into account when studying the relation between this and another transition system. A non-idle transition in system one now is similar to a sequence of transitions in system two if the following holds: this sequence of transitions in system two should consist of one non-idle transition and otherwise idle transitions, and the non-idle transition in this sequence should be similar to the transition in system one, i.e., the relevant elements of the configurations involved, should match.

In the context of our transition systems Trans_o and Trans_m, we can now phrase the following bisimulation lemma.

Lemma 6.1 *(weak bisimulation)* Let $+^*$ abbreviate $(execute \ + \ apply(\rho_1) \ + \ \ldots \ + \ apply(\rho_n))$. Let $\mathsf{Trans}_m(P)$ be the restriction of Trans_m to those transitions that are part of some sequence of transitions starting in initial meta-configuration $\langle P, s_0 \rangle$, with $s_0 \in S$ an arbitrary mental state and let $t \in \{execute, apply(\rho) \mid \rho \in \mathsf{Rule}\}$. Then a weak bisimulation exists between Trans_o and $\mathsf{Trans}_m(\mathsf{interpreter})$, i.e., the following properties hold.

$$s \rightarrow_t s' \text{ is a transition in } \mathsf{Trans}_o \Rightarrow_1$$
$$\langle \mathsf{interpreter}, s \rangle \rightarrow_{idle} \langle +^*; \mathsf{interpreter}, s \rangle \rightarrow_t \langle \mathsf{interpreter}, s' \rangle$$
$$\text{is a transition in } \mathsf{Trans}_m(\mathsf{interpreter})$$

$$\langle +^*; \mathsf{interpreter}, s \rangle \rightarrow_t \langle \mathsf{interpreter}, s' \rangle$$
$$\text{is a transition in } \mathsf{Trans}_m(\mathsf{interpreter}) \Rightarrow_2$$
$$s \rightarrow_t s' \text{ is a transition in } \mathsf{Trans}_o$$

Proof: ($\Rightarrow_1$) Assume $s \rightarrow_t s'$ is a transition in Trans_o for $t \in \{execute, apply(\rho) \mid \rho \in \mathsf{Rule}\}$. Using Proposition 6.1, the following then is a transition in Trans_m.

$$\langle (execute \ + \ apply(\rho_1) \ + \ \ldots \ + \ apply(\rho_n)); \mathsf{interpreter}, s \rangle \rightarrow_t$$
$$\langle \mathsf{interpreter}, s' \rangle \quad (6.1)$$

Furthermore, by the assumption that $s \rightarrow_t s'$ is a transition in Trans_o and by the fact that no reactive rules are contained in Rule (see Remark 6.1), we know that the plan of s is not empty as both rule application and basic action execution require a non-empty plan. Now, using the fact that the plan of s is not empty, the following transition can be derived in $\mathsf{Trans}_m(\mathsf{interpreter})$.

$$\langle \mathsf{interpreter}, s \rangle \rightarrow_{idle}$$
$$\langle (execute \ + \ apply(\rho_1) \ + \ \ldots \ + \ apply(\rho_n)); \mathsf{interpreter}, s \rangle \quad (6.2)$$

The transitions (6.2) and (6.1) can be concatenated, yielding the desired result.

($\Rightarrow_2$) Assume $\langle +^*; \mathsf{interpreter}, s\rangle \rightarrow_t \langle \mathsf{interpreter}, s'\rangle$ is a transition in $\mathsf{Trans}_m(\mathsf{interpreter})$. Then, $\langle +^*, s\rangle \rightarrow_t \langle E, s'\rangle$ must be a transition in Trans_m (Definition 6.14). Therefore, by Proposition 6.1, we can conclude that $s \rightarrow_t s'$ is a transition in Trans_o. □

We are now in a position to give the equivalence theorem of this section.

Theorem 6.1 *($\mathcal{O}_o = \mathcal{O}_m(\mathsf{interpreter})$)*

$$\forall s \in S : \mathcal{O}_o(s) = \mathcal{O}_m(\mathsf{interpreter})(s)$$

Proof: As equivalence of object-level and meta-level operational semantics holds for input state $\perp$ by Definition 6.18, we will only need to prove equivalence for input states $s \in S$. Proving this theorem amounts to showing the following: $s \in \mathcal{O}_o \Leftrightarrow s \in \mathcal{O}_m(\mathsf{interpreter})$.
($\Rightarrow$) Assume $s \in \mathcal{O}_o$. This means that a sequence of transitions $s_0 \rightarrow_{t_1} \ldots \rightarrow_{t_n} s$ must be derivable in Trans_o. By repeated application of Lemma 6.1, we know that then there must also be a sequence of transitions in $\mathsf{Trans}_m(\mathsf{interpreter})$ of the following form:

$$\begin{aligned}\langle \mathsf{interpreter}, s_0\rangle &\rightarrow_{idle} \ldots \rightarrow_{t_{n-1}} \\ &\langle \mathsf{interpreter}, s'\rangle \rightarrow_{idle} \langle +^*; \mathsf{interpreter}, s'\rangle \rightarrow_{t_n} \langle \mathsf{interpreter}, s\rangle. \quad (6.3)\end{aligned}$$

As $s \in \mathcal{O}_o$, we know that there cannot be a transition $s \rightarrow_{t_{n+1}} s''$ for some mental state s'', i.e., it is not possible to execute an *execute* or *apply* meta-action in s. Therefore, we know that the only possible transition from $\langle \mathsf{interpreter}, s\rangle$ in (6.3) above, is $\ldots \rightarrow_{idle} \langle E, s\rangle$. From this, we have that $s \in \mathcal{O}_m(\mathsf{interpreter})$.

($\Leftarrow$) Assume that $s \in \mathcal{O}_m(\mathsf{interpreter})$. Then there must be a sequence of transitions in $\mathsf{Trans}_m(\mathsf{interpreter})$ of the form:

$$\begin{aligned}\langle \mathsf{interpreter}, s_0\rangle &\rightarrow_{idle} \langle +^*; \mathsf{interpreter}, s_0\rangle \rightarrow_{t_1} \ldots \rightarrow_{t_{n-1}} \\ &\langle \mathsf{interpreter}, s'\rangle \rightarrow_{idle} \langle +^*; \mathsf{interpreter}, s'\rangle \rightarrow_{t_n} \\ &\langle \mathsf{interpreter}, s\rangle \rightarrow_{idle} \langle E, s\rangle.\end{aligned}$$

From this, we can conclude by Lemma 6.1 that $s_0 \rightarrow_{t_1} \ldots \rightarrow_{t_{n-1}} s' \rightarrow_{t_n} s \not\rightarrow$ must be a sequence of transitions in Trans_o. Therefore, it must be the case that $s \in \mathcal{O}_o$. □

Note that it is easy to show that $\mathcal{O}_o = \mathcal{O}_m(P)$ does not hold for all meta-programs P.

6.4 Denotational Semantics

In this section, we will define the denotational semantics of meta-programs. The method used is the fixed point approach as can be found in [Stoy, 1977]. The semantics greatly resembles the one in [de Bakker, 1980, Chapter 7] to which we refer for a detailed explanation of the subject.

A denotational semantics for a programming language in general is, like an operational semantics, a function taking a statement P and a state s and yielding a state (or set of states in case of a non-deterministic language) resulting from executing P in s. The denotational semantics for meta-programs is thus, like the operational semantics of Definition 6.18, a function taking a meta-program P and mental state s and yielding the set of mental states resulting from executing P in s, i.e., a function of type $Prog \rightarrow (S_\perp \rightarrow \wp(S_\perp))$[2]. Contrary however to an operational semantic function, a denotational semantic function is not defined using the concept of computation sequences and, in contrast with most operational semantics, it *is* defined compositionally [Tennent, 1991, Mosses, 1990, de Bakker, 1980].

6.4.1 Preliminaries

In order to define the denotational semantics of meta-programs, we need some mathematical machinery. Most importantly, the domains used in defining the semantics of meta-programs are designed as so-called complete partial orders (CPOs). A CPO is a set with an ordering on its elements with certain characteristics. This concept is defined in terms of the notions of partially ordered sets, least upper bounds and chains (see also [de Bakker, 1980] for a rigorous treatment of the subject).

Definition 6.20 *(partially ordered set)* Let C be an arbitrary set. A partial order $\sqsubseteq$ on C is a subset of $C \times C$ which satisfies:

1. $c \sqsubseteq c$ (reflexivity),
2. if $c_1 \sqsubseteq c_2$ and $c_2 \sqsubseteq c_1$ then $c_1 = c_2$ (antisymmetry),
3. if $c_1 \sqsubseteq c_2$ and $c_2 \sqsubseteq c_3$ then $c_1 \sqsubseteq c_3$ (transitivity).

In the sequel, we will be concerned not only with arbitrary sets with partial orderings, but also with sets of functions with an ordering. A partial ordering on a set of functions of type $C_1 \rightarrow C_2$ can be derived from the orderings on C_1 and C_2 as defined below.

[2]The type of the denotational semantic function is actually slightly different as will become clear in the sequel, but that is not important for the current discussion.

Definition 6.21 *(partial ordering on functions)* Let $(C_1, \sqsubseteq_1)$ and $(C_2, \sqsubseteq_2)$ be two partially ordered sets. An ordering $\sqsubseteq$ on $C_1 \to C_2$ is defined as follows, where $f, g \in C_1 \to C_2$:

$$f \sqsubseteq g \Leftrightarrow \forall c \in C_1 : f(c) \sqsubseteq_2 g(c).$$

Definition 6.22 *(least upper bound)* Let $C' \subseteq C$. $z \in C$ is called the least upper bound of C' if:

1. z is an upper bound: $\forall x \in C' : x \sqsubseteq z$,
2. z is the *least* upper bound: $\forall y \in C : ((\forall x \in C' : x \sqsubseteq y) \Rightarrow z \sqsubseteq y)$.

The least upper bound of a set C' will be denoted by $\bigsqcup C'$.

Definition 6.23 *(least upper bound of a sequence)* The least upper bound of a sequence $\langle c_0, c_1, \ldots \rangle$ is denoted by $\bigsqcup_{i=0}^{\infty} c_i$ or by $\bigsqcup \langle c_i \rangle_{i=0}^{\infty}$ and is defined as follows, where "c in $\langle c_i \rangle_{i=0}^{\infty}$" means that c is an element of the sequence $\langle c_i \rangle_{i=0}^{\infty}$:

$$\bigsqcup \langle c_i \rangle_{i=0}^{\infty} = \bigsqcup \{c \mid c \text{ in } \langle c_i \rangle_{i=0}^{\infty}\}.$$

Definition 6.24 *(chains)* A chain on $(C, \sqsubseteq)$ is an infinite sequence $\langle c_i \rangle_{i=0}^{\infty}$ such that for $i \in \mathbb{N} : c_i \sqsubseteq c_{i+1}$.

Having defined partially ordered sets, least upper bounds and chains, we are now in a position to define complete partially ordered sets.

Definition 6.25 *(CPO)* A complete partially ordered set is a set C with a partial order $\sqsubseteq$ which satisfies the following requirements:

1. there is a least element with respect to $\sqsubseteq$, i.e., an element $\bot \in C$ such that $\forall c \in C : \bot \sqsubseteq c$,
2. each chain $\langle c_i \rangle_{i=0}^{\infty}$ in C has a least upper bound $(\bigsqcup_{i=0}^{\infty} c_i) \in C$.

The following facts about CPOs of functions will turn out to be useful. For proofs, see for example [de Bakker, 1980].

Fact 6.1 *(CPO of functions)* Let $(C_1, \sqsubseteq_1)$ and $(C_2, \sqsubseteq_2)$ be CPOs. Then $(C_1 \to C_2, \sqsubseteq)$ with $\sqsubseteq$ as in Definition 6.21 is a CPO.

Fact 6.2 *(least upper bound of a chain of functions)* Let $(C_1, \sqsubseteq_1)$ and $(C_2, \sqsubseteq_2)$ be CPOs and let $\langle f_i \rangle_{i=0}^{\infty}$ be a chain of functions in $C_1 \to C_2$. Then the function $\lambda c_1 \cdot \bigsqcup_{i=0}^{\infty} f_i(c_1)$ is the least upper bound of this chain and therefore $(\bigsqcup_{i=0}^{\infty} f_i)(c_1) = \bigsqcup_{i=0}^{\infty} f_i(c_1)$ for all $c_1 \in C_1$.

The semantics of meta-programs will be defined using the notion of the least fixed point of a function on a CPO.

Definition 6.26 *(least fixed point)* Let $(C, \sqsubseteq)$ a CPO, $f : C \to C$ and let $x \in C$.

- x is a fixed point of f iff $f(x) = x$
- x is a least fixed point of f iff x is a fixed point of f and for each fixed point y of f: $x \sqsubseteq y$

The least fixed point of a function f is denoted by μf.

Finally, we will need the following definition and fact.

Definition 6.27 *(continuity)* Let $(C_1, \sqsubseteq_1), (C_2, \sqsubseteq_2)$ be CPOs. Then a function $f : C_1 \to C_2$ is continuous iff for each chain $\langle c_i \rangle_{i=0}^{\infty}$ in C_1, the following holds:

$$f(\bigsqcup_{i=0}^{\infty} c_i) = \bigsqcup_{i=0}^{\infty} f(c_i).$$

Fact 6.3 *(fixed point theorem)* Let C be a CPO and let $f : C \to C$. If f is continuous, then the least fixed point μf exists and equals $\bigsqcup_{i=0}^{\infty} f^i(\bot)$, where $f^0(\bot) = \bot$ and $f^{i+1}(\bot) = f(f^i(\bot))$.

For a proof, see for example De Bakker [de Bakker, 1980].

6.4.2 Definition of Meta-Level Denotational Semantics

We will now show how the domains used in defining the semantics of meta-programs are designed as CPOs. The reason for designing these as CPOs will become clear in the sequel.

Definition 6.28 *(domains of interpretation)* Let W be the set of truth values of Definition 6.12 and let S be the set of possible mental states of Definition 6.5. Then the sets $W_\bot$ and $S_\bot$ are defined as CPOs as follows:

$$\begin{array}{lcll} W_\bot & = & W \cup \{\bot_{W_\bot}\} & \text{CPO by } \beta_1 \sqsubseteq \beta_2 \text{ iff } \beta_1 = \bot_{W_\bot} \text{ or } \beta_1 = \beta_2, \\ S_\bot & = & S \cup \{\bot\} & \text{CPO analogously.} \end{array}$$

Note that we use $\bot$ to denote the bottom element of $S_\bot$ and that we use $\bot_C$ for the bottom element of any other set C. As the set of mental states is extended with a bottom element, we extend the semantics of boolean expressions of Definition 6.12 to a strict function, i.e., yielding $\bot_{W_\bot}$ for an input state $\bot$.

In the definition of the denotational semantics, we will use an if-then-else function as defined below.

Definition 6.29 *(if-then-else)* Let C be a CPO, $c_1, c_2, \bot_C \in C$ and $\beta \in W_\bot$. Then the if-then-else function of type $W_\bot \to C$ is defined as follows.

$$\texttt{if } \beta \texttt{ then } c_1 \texttt{ else } c_2 \texttt{ fi} = \begin{cases} c_1 & \text{if } \beta = tt \\ c_2 & \text{if } \beta = ff \\ \bot_C & \text{if } \beta = \bot_{W_\bot} \end{cases}$$

Because our meta-language is non-deterministic, the denotational semantics is not a function from states to states, but a function from states to *sets of states*. These resulting sets of states can be finite or infinite. In case of bounded non-determinism[3], these infinite sets of states have $\perp$ as one of their members. This property may be explained by viewing the execution of a program as a tree of computations and then using König's lemma which tells us that a finitely-branching tree with infinitely many nodes has at least one infinite path (see [de Bakker, 1980]). The meta-language is indeed bounded non-deterministic[4], and the result of executing a meta-program P in some state is thus either a finite set of states or an infinite set of states containing $\perp$. We therefore specify the following domain as the result domain of the denotational semantic function instead of $\wp(S_\perp)$.

Definition 6.30 *(T)* The set T with typical element τ is defined as follows: $T = \{\tau \in \wp(S_\perp) \mid \tau \text{ finite or } \perp \in \tau\}$.

The advantage of using T instead of $\wp(S_\perp)$ as the result domain is that T can nicely be designed as a CPO with the following ordering [Egli, 1975].

Definition 6.31 *(Egli-Milner ordering)* Let $\tau_1, \tau_2 \in T$. $\tau_1 \sqsubseteq \tau_2$ holds iff either $\perp \in \tau_1$ and $\tau_1 \setminus \{\perp\} \subseteq \tau_2$, or $\perp \notin \tau_1$ and $\tau_1 = \tau_2$. Under this ordering, the set $\{\perp\}$ is $\perp_T$.

We are now ready to give the denotational semantics of meta-programs. We will first give the definition and then justify and explain it.

Definition 6.32 *(denotational semantics of meta-programs)* Let $\phi_1, \phi_2 : S_\perp \to T$. Then we define the following functions.

$$\begin{array}{lllll} \hat{\phi} & : T \to T & = & \lambda\tau \cdot \bigcup_{s\in\tau} \phi(s) \\ \phi_1 \circ \phi_2 & : S_\perp \to T & = & \lambda s \cdot \hat{\phi_1}(\phi_2(s)) \end{array}$$

Let $(\pi, \sigma) \in S$. The denotational semantics of meta-programs $\mathcal{M} : Prog \to (S_\perp \to T)$ is then defined as follows.

$$\begin{array}{lcl} \mathcal{M}[\![execute]\!](\pi,\sigma) & = & \begin{cases} \{(\pi', \sigma')\} & \text{if } \pi = a;\pi' \\ & \text{with } a \in \mathsf{BasicAction} \text{ and} \\ & \mathcal{T}(a,\sigma) = \sigma' \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{M}[\![execute]\!]\ \perp & = & \perp_T \\ \mathcal{M}[\![apply(\rho)]\!](\pi,\sigma) & = & \begin{cases} \{(\pi_b \bullet \pi', \sigma)\} & \text{if } \sigma \models \psi \text{ and } \pi = \pi_h \bullet \pi' \\ & \text{with } \rho : \pi_h \mid \psi \rightsquigarrow \pi_b \in \mathsf{Rule} \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{M}[\![apply(\rho)]\!]\ \perp & = & \perp_T \end{array}$$

[3]Bounded non-determinism means that at any state during computation, the number of possible next states is finite.

[4]Only a finite number of rule applications and action executions are possible in any state, since the set of plan revision rules of an agent is finite (see Definition 6.4).

$$\begin{array}{lcl} \mathcal{M}[\![\texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}]\!] & = & \mu\Phi \\ \mathcal{M}[\![P_1; P_2]\!] & = & \mathcal{M}[\![P_2]\!] \circ \mathcal{M}[\![P_1]\!] \\ \mathcal{M}[\![P_1 + P_2]\!] & = & \mathcal{M}[\![P_1]\!] \cup \mathcal{M}[\![P_2]\!] \end{array}$$

The function $\Phi : (S_\perp \to T) \to (S_\perp \to T)$ used above is defined as $\lambda\phi \cdot \lambda s \cdot \texttt{if}\ \mathcal{W}(b)(s)\ \texttt{then}\ \hat{\phi}(\mathcal{M}[\![P]\!](s))\ \texttt{else}\ \{s\}\ \texttt{fi}$, using Definition 6.29.

Meta-actions

The semantics of meta-actions is straightforward. The result of executing an *execute* meta-action in some mental state s is a set containing the mental state resulting from executing the basic action of the plan of s. The result is empty if there is no basic action on the plan to execute. The result of executing an $apply(\rho)$ meta-action in state s is a set containing the mental state resulting from applying ρ in s. If ρ is not applicable, the result is the empty set.

While

The semantics of the `while` construct is more involved, but based on standard techniques as described in [de Bakker, 1980]. In summary, it is as follows. What we want to do, is define a function specifying the semantics of the `while` construct $\mathcal{M}[\![\texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}]\!]$, the type of which should be $S_\perp \to T$, in accordance with the type of $\mathcal{M}$. The function should be defined compositionally, i.e., it can only use the semantics of the guard and of the body of the `while`. This is required for $\mathcal{M}$ to be well-defined. The requirement of compositionality is satisfied, as the semantics is defined to be the least fixed point of the operator Φ, which is defined in terms of the semantics of the guard and body of the `while`.

The least fixed point of an operator does not always exist. By the fixed point theorem however (Fact 6.3), we know that if the operator is continuous (Definition 6.27), the least fixed point *does* exist and is obtainable within ω steps. By proving that Φ is continuous, we can thus conclude that $\mu\Phi$ exists and therefore that $\mathcal{M}$ is well-defined.

We explain the semantics in detail by first defining a set of requirements on the semantics and then giving an intuitive understanding of the general ideas behind the semantics. Next, we will go into more detail on the semantics of a `while` construct in deterministic languages, followed by details on the semantics as defined above for our non-deterministic language.

Now, what we want to do, is define a function specifying the semantics of the `while` construct $\mathcal{M}[\![\texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}]\!]$, the type of which should be $S_\perp \to T$, in accordance with the type of $\mathcal{M}$. The function can moreover not be defined circularly, but it should be defined compositionally, i.e., it can only use the semantics of the guard and of the body of the `while`. This ensures that $\mathcal{M}$ is well-defined. The semantics of the `while` can thus not be defined as $\mathcal{M}[\![\texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}]\!] = \mathcal{M}[\![\texttt{if}\ b\ \texttt{then}\ P; \texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}\ \texttt{else}\ \mathit{skip}\ \texttt{fi}]\!]$ where

skip is a statement doing nothing, because this would violate the requirement of compositionality.

Intuitively, the semantics of `while` b `do` P `od` should correspond to repeatedly executing P, until b is false. The semantics could thus be something like:

$$\mathcal{M}[\![\texttt{if } b \texttt{ then } P; \texttt{if } b \texttt{ then } P; \ldots \texttt{ else } \mathit{skip} \texttt{ fi else } \mathit{skip} \texttt{ fi}]\!].$$

The number of nestings of if-then-else constructs should however be infinite, as we cannot determine in advance how many times the body of the while loop will be executed, worse still, it could be the case that it will be executed an infinite number of times in case of non-termination. As programs are however by definition finite syntactic objects (see Definition 6.6), the semantics of the `while` cannot be defined in this way. The idea of the solution now is, *not* to try to specify the semantics *at once* using infinitely many nestings of if-then-else constructs, but instead to specify the semantics using approximating functions, where some approximation is at least as good as another if it contains more nestings of if-then-else constructs. So to be a little more specific, what we do is specify a sequence of approximating functions $\langle \phi_i \rangle_{i=0}^{\infty}$, where ϕ_i roughly speaking corresponds to executing the body of the `while` construct less than i times and the idea now is that the limit of this sequence of approximations is the semantics of the `while` we are looking for.

Deterministic languages Having provided some vague intuitive ideas about the semantics, we will now go into more detail and use the theory on CPOs of Section 6.4.1. To simplify matters, we will first consider a deterministic language. The function defining the semantics of the `while` should then be of type $S_\perp \to S_\perp$. The domain $S_\perp$ is designed as a CPO and therefore the domain of functions $S_\perp \to S_\perp$ is also a CPO (see Fact 6.1). What we are looking for, is a chain of approximating functions $\langle \phi_i \rangle_{i=0}^{\infty}$ in this CPO $S_\perp \to S_\perp$, the least upper bound of which should yield the desired semantics for the `while`. As we are looking for a chain (see Definition 6.24) of functions $\langle \phi_i \rangle_{i=0}^{\infty}$, it should be the case that for all $s \in S_\perp : \phi_i(s) \sqsubseteq \phi_j(s)$ for $i \leq j$. Intuitively, a function ϕ_j is "as least as good" an approximation as ϕ_i.

Approximating functions with the following behavior will do the trick. A function ϕ_i should be defined such, that the result of ϕ_i applied to an initial state s_0, is the state in which the while loop terminates if less than i runs through the loop are needed for termination. If i or more runs are needed, i.e., if the execution of the `while` has not terminated after $i-1$ runs, the result should be $\perp$. To illustrate how these approximating functions can be defined to yield this behavior, we will give the first few approximations ϕ_0, ϕ_1 and ϕ_2. In order to do this, we need to introduce a special statement *diverge*, which yields the state $\perp$ if executed. It can be shown that with the functions ϕ_i as defined below, $\langle \phi_i \rangle_{i=0}^{\infty}$ is indeed a chain. Therefore it has a least upper bound by definition,

which ensures that the semantics is well-defined.

$$\begin{array}{rcl}
\phi_0 & = & \mathcal{M}[\![\mathit{diverge}]\!] \\
\\
\phi_1 & = & \mathcal{M}[\![\texttt{if } b \texttt{ then } P; \mathit{diverge} \texttt{ else } \mathit{skip} \texttt{ fi}]\!] \\
\phi_2 & = & \mathcal{M}[\![\texttt{if } b \texttt{ then } \\
 & & \qquad P; \texttt{if } b \texttt{ then } P; \mathit{diverge} \texttt{ else } \mathit{skip} \texttt{ fi} \\
 & & \texttt{else } \mathit{skip} \texttt{ fi}]\!] \\
 & \vdots &
\end{array}$$

These definitions can be generalized, yielding the following definition of the functions ϕ_i in which we use the if-then-else function of Definition 6.29.

$$\begin{array}{rcl}
\phi_0 & = & \bot_{S_\bot \to S_\bot} \\
 & = & \lambda s \cdot \bot \\
\phi_{i+1} & = & \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \phi_i(\mathcal{M}[\![P]\!](s)) \texttt{ else } s \texttt{ fi}
\end{array}$$

Now, as $\langle \phi_i \rangle_{i=0}^{\infty}$ is a chain, so is $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$. Using Fact 6.2, we know that $\bigsqcup_{i=0}^{\infty} \phi_i = \lambda s_0 \cdot \bigsqcup_{i=0}^{\infty} \phi_i(s_0)$ or $(\bigsqcup_{i=0}^{\infty} \phi_i)(s_0) = \bigsqcup_{i=0}^{\infty} \phi_i(s_0)$, i.e., the semantics of the execution of the `while` construct in an initial state s_0, is the least upper bound of the chain $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$.

The semantic function defined in this way, will indeed yield the desired behavior for the while loop as sketched above, which we will show now. If the while loop is non-terminating, the chain $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$ will be $\langle \bot, \bot, \bot, \ldots \rangle$. This is because the guard b will remain true and therefore the statement *diverge* will be "reached" in each ϕ_i. Taking the least upper bound of this chain will give us $\bot$, which corresponds to the desired semantics of non-terminating while loops. If the loop terminates in some state s, the sequence $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$ will be a chain of the form $\langle \bot, \bot, \bot, \ldots, \bot, s, s, s, \ldots \rangle$, as can easily be checked. The state s then is the least upper bound.

Finally, to prepare for the treatment of the semantics as we have defined it for the non-deterministic case, the following must still be explained. Above, we have defined $\mathcal{M}[\![\texttt{while } b \texttt{ do } P \texttt{ od}]\!]$ as $\bigsqcup_{i=0}^{\infty} \phi_i$. Instead of defining the semantics using least upper bounds, we could have given an equivalent least fixed point characterization as follows. Let $\phi = \bigsqcup_{i=0}^{\infty} \phi_i$. Then we can give an operator $\Phi : (S_\bot \to S_\bot) \to (S_\bot \to S_\bot)$, i.e., a function on CPO $S_\bot \to S_\bot$, such that the least fixed point of this operator equals ϕ, i.e., $\mu\Phi = \phi$. This operator Φ is the function $\lambda\phi \cdot \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \phi(\mathcal{M}[\![P]\!](s)) \texttt{ else } s \texttt{ fi}$. We know that if Φ is continuous, $\mu\Phi = \bigsqcup_{i=0}^{\infty} \Phi^i(\bot_{S_\bot \to S_\bot})$ by the least fixed point theorem (Fact 6.3). It can be shown that Φ is indeed continuous and furthermore, that $\Phi^i(\bot_{S_\bot \to S_\bot}) = \phi_i$. Therefore $\bigsqcup_{i=0}^{\infty} \phi_i = \bigsqcup_{i=0}^{\infty} \Phi^i(\bot_{S_\bot \to S_\bot})$ and thus $\phi = \mu\Phi$. The question of whether to specify the semantics of a `while` construct using least upper bounds or least fixed points, is basically a matter of taste. We have chosen to use a least fixed point characterization in the semantics of meta-programs.

Non-deterministic languages Having explained the denotational semantics of a while loop in deterministic languages, we will now move on to the non-deterministic case, as our meta-programming language is non-deterministic. In the non-deterministic case, the execution of a while loop could lead to a set of possible resulting end states, including the state $\perp$ if there is a possibility of non-termination. A certain approximation ϕ_i now is a function yielding a set of end states that can be reached in less than i runs through the loop. It will contain bottom if it is possible that the execution of the loop has not terminated after i runs. The limit of the sequence $\langle \phi_i(s_0) \rangle_{i=0}^{\infty}$ will thus be either a finite set of states possibly containing $\perp$ (if there is a possibility of non-termination) or an infinite set which will always contain $\perp$ because of Königs lemma (see introduction). The semantic function we are looking for, will thus be of type $S_\perp \to T$.

As stated, the semantics of the `while` construct in our meta-language is defined using least fixed points. To be more specific, it is defined as the least fixed point of the operator $\Phi : (S_\perp \to T) \to (S_\perp \to T)$ (see Definition 6.32). Φ is thus a function on the CPO $S_\perp \to T$ (Definitions 6.28 and 6.31, and Fact 6.1) and the semantics of the `while` is defined to be $\mu\Phi$. We must make sure that $\mu\Phi$ actually exists, in order for $\mathcal{M}$ to be well-defined. We do this by showing that Φ is continuous (see Section 6.4.3), in which case $\mu\Phi = \bigsqcup_{i=0}^{\infty} \Phi^i(\perp_{S_\perp \to T})$. The bottom element $\perp_{S_\perp \to T}$ of the CPO $S_\perp \to T$ is $\lambda s \cdot \{\perp\}$, i.e., a function that takes some state and returns a set of states containing only the bottom state. Note that the type of Φ is such, that $\mu\Phi$ yields a semantic function $\phi : S_\perp \to T$, corresponding to the type of the function $\mathcal{M}[\![\texttt{while}\ b\ \texttt{do}\ P\ \texttt{od}]\!]$. The operator Φ is thus of the desired type.

The operator Φ we use, is a non-deterministic version of the Φ operator of the previous paragraph. This is established through the function $\hat{} : (S_\perp \to T) \to (T \to T)$. This function takes a function ϕ of type $S_\perp \to T$ and a set $\tau \in T$ and returns the union of ϕ applied to each element $s \in \tau$, i.e., $\bigcup_{s \in \tau} \phi(s)$. We will now give the first few elements of the sequence of approximations $\langle \Phi^i(\perp_{S_\perp \to T}) \rangle_{i=0}^{\infty}$, to illustrate how Φ is defined in the non-deterministic case. For reasons of presentation, we will assume that $\mathcal{M}[\![P]\!](s) \neq \emptyset$ in which case $(\hat{}\,(\lambda s \cdot \{\perp\}))(\mathcal{M}[\![P]\!](s)) = \{\perp\}$. As it follows from the definition of $\hat{\phi}$ (Definition 6.32) that $\hat{\phi}(\emptyset) = \emptyset$, some equivalences as stated below would not

hold in this case.

$$
\begin{array}{lcl}
\Phi^0(\bot_{S_\bot \to T}) & = & \lambda s \cdot \{\bot\} \\
\Phi^1(\bot_{S_\bot \to T}) & = & \Phi(\Phi^0(\bot_{S_\bot \to T})) \\
 & = & \Phi(\lambda s \cdot \{\bot\}) \\
 & = & \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } (\hat{}(\lambda s \cdot \{\bot\}))(\mathcal{M}[\![P]\!](s)) \texttt{ else } \{s\} \texttt{ fi} \\
 & = & \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \{\bot\} \texttt{ else } \{s\} \texttt{ fi} \\
\Phi^2(\bot_{S_\bot \to T}) & = & \Phi(\Phi^1(\bot_{S_\bot \to T})) \\
 & = & \Phi(\lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \{\bot\} \texttt{ else } \{s\} \texttt{ fi}) \\
 & = & \lambda s \cdot \texttt{if } \mathcal{W}(b)(s) \\
 & & \quad \texttt{then} \\
 & & \quad\quad (\hat{}(\lambda s' \cdot \texttt{if } \mathcal{W}(b)(s') \\
 & & \quad\quad\quad\quad \texttt{then} \\
 & & \quad\quad\quad\quad\quad \{\bot\} \\
 & & \quad\quad\quad\quad \texttt{else} \\
 & & \quad\quad\quad\quad\quad \{s'\} \texttt{ fi}))(\mathcal{M}[\![P]\!](s)) \\
 & & \quad \texttt{else } \{s\} \texttt{ fi} \\
\Phi^3(\bot_{S_\bot \to T}) & = & \Phi(\Phi^2(\bot_{S_\bot \to T})) \\
 & = & \ldots
\end{array}
$$

The zero$^{\text{th}}$ approximation by definition (see Fact 6.3) always yields the bottom element of the CPO $S_\bot \to T$. The first approximation is a function that takes an initial state s and yields either the set $\{\bot\}$ if the guard is true in s, i.e., if there will be one or more runs through the loop, or the set $\{s\}$ if the guard is false in s, i.e., if the while loop is such that it terminates in s without going through the loop. The second approximation is a function that takes an initial state s and yields, like the first approximation, the set $\{s\}$ if the guard is false in s. It thus returns the same result as the first approximation if it takes less than one runs through the loop to terminate. This is exactly what we want, as the first approximation will be as good as it gets in this case. If the guard is true in s, the function $\lambda s' \cdot \texttt{if } \mathcal{W}(b)(s') \texttt{ then } \{\bot\} \texttt{ else } \{s'\} \texttt{ fi}$ is applied to each state in the set of states resulting from executing the body of the while in s, i.e., the set $\mathcal{M}[\![P]\!](s)$ which we will refer to by τ'. The function thus takes some state s' from τ' and either yields $\{s'\}$ if the guard is true in s', i.e., if the while can terminate in s' after one run through the loop, or yields $\{\bot\}$ if the guard is false in s', i.e., if the execution path going through s' has not ended. The function $\hat{}$ now takes the union of the results for each s', yielding a set of states containing the states in which the while loop can end after one run through the loop, and containing $\bot$ if it is possible that the while loop has not terminated after one run. The function Φ is thus defined such that a certain approximation $\Phi^i(\bot_{S_\bot \to T})$ is a function yielding a set of end states that can be reached in less than i runs through the loop. It will contain $\bot$ if it is possible that the execution of the loop has not terminated after i runs.

Sequential Composition and Non-Deterministic Choice

The semantics of the sequential composition and non-deterministic choice operator is as one would expect.

6.4.3 Continuity of Φ

In this section, we prove continuity of the function Φ of Definition 6.32, because, if this is the case, the least fixed point $\mu\Phi$ exists and is obtainable within ω steps. This is necessary in order for $\mathcal{M}$ to be well-defined.

Theorem 6.2 *(continuity of Φ)* The function Φ as given in Definition 6.32 is continuous.

The proof is analogous to continuity proofs given in [de Bakker, 1980], and uses two lemmas and a fact. In Definition 6.27, the concept of continuity was defined. As we will state below in Fact 6.4, an equivalent definition can be given using the concept of monotonicity of a function.

Definition 6.33 *(monotonicity)* Let $(C, \sqsubseteq), (C', \sqsubseteq)$ be CPOs and $c_1, c_2 \in C$. Then a function $f : C \to C'$ is monotone iff the following holds:

$$c_1 \sqsubseteq c_2 \Leftrightarrow f(c_1) \sqsubseteq f(c_2).$$

Fact 6.4 *(continuity)* Let $(C, \sqsubseteq), (C', \sqsubseteq)$ be CPOs and let $f : C \to C'$ be a function. Then the following holds.

$$\begin{array}{c} \text{for all chains } \langle c_i \rangle_{i=0}^{\infty} \text{ in } C : f(\bigsqcup_{i=0}^{\infty} c_i) = \bigsqcup_{i=0}^{\infty} f(c_i) \\ \Leftrightarrow \\ f \text{ is monotone and for all chains } \langle c_i \rangle_{i=0}^{\infty} \text{ in } C : f(\bigsqcup_{i=0}^{\infty} c_i) \sqsubseteq \bigsqcup_{i=0}^{\infty} f(c_i) \end{array}$$

Proof: This proof is an adaptation of a proof provided to us by Paul Harrenstein.

($\Rightarrow$): Assume for all chains $\langle c_i \rangle_{i=0}^{\infty}$ in C : $f(\bigsqcup \langle c_i \rangle_{i=0}^{\infty}) = \bigsqcup \langle f(c_i) \rangle_{i=0}^{\infty}$. Then $f(\bigsqcup \langle c_i \rangle_{i=0}^{\infty}) \sqsubseteq \bigsqcup \langle f(c_i) \rangle_{i=0}^{\infty}$ trivially holds for all chains $\langle c_i \rangle_{i=0}^{\infty}$ in C. To prove: monotonicity of f.
Let $c, c' \in C$ and assume $c \sqsubseteq c'$. The following holds: $f(c) \sqsubseteq \bigsqcup \{f(c), f(c')\}$. As $\bigsqcup \{f(c), f(c')\} = \bigsqcup \langle f(c_i) \rangle_{i=0}^{\infty}$ with $\langle f(c_i) \rangle_{i=0}^{\infty} \in Chain(\{f(c), f(c')\})$ where $Chain(\{f(c), f(c')\})$ is the set of chains that can be formed using the elements of the set $\{f(c), f(c')\}$, we can conclude that $f(c) \sqsubseteq f(\bigsqcup \langle c_i \rangle_{i=0}^{\infty})$ with $\langle c_i \rangle_{i=0}^{\infty} \in Chain(\{c, c'\})$ by assumption. As $\bigsqcup \langle c_i \rangle_{i=0}^{\infty} = \bigsqcup \{c, c'\}$, we can conclude that $f(c) \sqsubseteq f(c')$, using that $c' = \bigsqcup \{c, c'\}$.

($\Leftarrow$): Assume that f is monotone and that $f(\bigsqcup \langle c_i \rangle_{i=0}^{\infty}) \sqsubseteq \bigsqcup \langle f(c_i) \rangle_{i=0}^{\infty}$

holds for all chains $\langle c_i \rangle_{i=0}^{\infty}$ in C. Then we need to prove that for all chains $\langle c_i \rangle_{i=0}^{\infty}$ in $C : \bigsqcup \langle f(c_i) \rangle_{i=0}^{\infty} \sqsubseteq f(\bigsqcup \langle c_i \rangle_{i=0}^{\infty})$. Take an arbitrary chain $\langle c_i \rangle_{i=0}^{\infty}$ in C. Let $X = \{c \mid c \text{ in } \langle c_i \rangle_{i=0}^{\infty}\}$ and let $X' = \{f(c) \mid c \text{ in } \langle c_i \rangle_{i=0}^{\infty}\} = \{f(x) \mid x \in X\}$. To prove: $\bigsqcup X' \sqsubseteq f(\bigsqcup X)$.
Take some $x \in X$. Then $x \sqsubseteq \bigsqcup X$ and thus by monotonicity of f: $f(x) \sqsubseteq f(\bigsqcup X)$. With x having been chosen arbitrarily, we may conclude that $f(\bigsqcup X)$ is an upper bound of X'. Hence, $\bigsqcup X' \sqsubseteq f(\bigsqcup X)$. □

Below, we will prove continuity of Φ by proving that Φ is monotone and that for all chains $\langle \phi_i \rangle_{i=0}^{\infty}$ in $S_\perp \to T$, the following holds: $\Phi(\bigsqcup_{i=0}^{\infty} \phi_i) \sqsubseteq \bigsqcup_{i=0}^{\infty} \Phi(\phi_i)$.

Lemma 6.2 *(monotonicity of Φ)* The function Φ as given in Definition 6.32 is monotone, i.e., the following holds for all $\phi_i, \phi_j \in S_\perp \to T$:

$$\phi_i \sqsubseteq \phi_j \Rightarrow \Phi(\phi_i) \sqsubseteq \Phi(\phi_j).$$

Proof: Take arbitrary $\phi_i, \phi_j \in S_\perp \to T$. Let $\phi_i \sqsubseteq \phi_j$. Then we need to prove that $\forall s \in S_\perp : \Phi(\phi_i)(s) \sqsubseteq \Phi(\phi_j)(s)$. Take an arbitrary $s \in S_\perp$. We need to prove that $\Phi(\phi_i)(s) \sqsubseteq \Phi(\phi_j)(s)$, i.e., that

$$\texttt{if } \mathcal{W}(b)(s) \texttt{ then } \hat{\phi}_i(\mathcal{M}[\![P]\!](s)) \texttt{ else } \{s\} \texttt{ fi } \sqsubseteq \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \hat{\phi}_j(\mathcal{M}[\![P]\!](s)) \texttt{ else } \{s\} \texttt{ fi}.$$

We distinguish three cases.

1. Let $\mathcal{W}(b)(s) = \perp_{W_\perp}$, then to prove: $\{\perp\} \sqsubseteq \{\perp\}$. This is true by Definition 6.31.
2. Let $\mathcal{W}(b)(s) = ff$, then to prove: $\{s\} \sqsubseteq \{s\}$. This is true by Definition 6.31.
3. Let $\mathcal{W}(b)(s) = tt$, then to prove: $\hat{\phi}_i(\mathcal{M}[\![P]\!](s)) \sqsubseteq \hat{\phi}_j(\mathcal{M}[\![P]\!](s))$. Let $\tau' = \mathcal{M}[\![P]\!](s)$. Using the definition of $\hat{\phi}$, we rewrite what needs to be proven into $\bigcup_{s' \in \tau'} \phi_i(s') \sqsubseteq \bigcup_{s' \in \tau'} \phi_j(s')$. Now we can distinguish two cases.
   (a) Let $\perp \notin \bigcup_{s' \in \tau'} \phi_i(s')$. Then to prove: $\bigcup_{s' \in \tau'} \phi_i(s') = \bigcup_{s' \in \tau'} \phi_j(s')$. From the assumption that $\perp \notin \bigcup_{s' \in \tau'} \phi_i(s')$, we can conclude that $\perp \notin \phi_i(s')$ for all $s' \in \tau'$. Using the assumption that $\phi_i(s) \sqsubseteq \phi_j(s)$ for all $s \in S_\perp$, we have that $\phi_i(s') = \phi_j(s')$ for all $s' \in \tau'$ and therefore $\bigcup_{s' \in \tau'} \phi_i(s') = \bigcup_{s' \in \tau'} \phi_j(s')$.
   (b) Let $\perp \in \bigcup_{s' \in \tau'} \phi_i(s')$. Then to prove: $(\bigcup_{s' \in \tau'} \phi_i(s')) \setminus \{\perp\} \subseteq \bigcup_{s' \in \tau'} \phi_j(s')$, i.e., $\bigcup_{s' \in \tau'} (\phi_i(s') \setminus \{\perp\}) \subseteq \bigcup_{s' \in \tau'} \phi_j(s')$. Using the assumption that $\phi_i(s) \sqsubseteq \phi_j(s)$ for all $s \in S_\perp$, we have that for all $s' \in \tau'$, either $\phi_i(s') \setminus \{\perp\} \subseteq \phi_j(s')$ or $\phi_i(s') = \phi_j(s')$, depending on whether $\perp \in \phi_i(s)$. From this we can conclude that $\bigcup_{s' \in \tau'} (\phi_i(s') \setminus \{\perp\}) \subseteq \bigcup_{s' \in \tau'} \phi_j(s')$.

□

As we now have that Φ is monotone, proving continuity comes down to proving the following lemma.

Lemma 6.3 For all chains $\langle \phi_i \rangle_{i=0}^{\infty}$ in $S_\perp \to T$, the following holds:

$$\Phi(\bigsqcup_{i=0}^{\infty} \phi_i) \sqsubseteq \bigsqcup_{i=0}^{\infty} \Phi(\phi_i).$$

Proof: We have to prove that for all chains $\langle \phi_i \rangle_{i=0}^{\infty}$ in $S_\perp \to T$ and for all $s \in S_\perp$, the following holds: $(\Phi(\bigsqcup_{i=0}^{\infty} \phi_i))(s) \sqsubseteq (\bigsqcup_{i=0}^{\infty} \Phi(\phi_i))(s)$. Take an arbitrary chain $\langle \phi_i \rangle_{i=0}^{\infty}$ in $S_\perp \to T$ and an arbitrary state $s \in S_\perp$. Then to prove:

$$\texttt{if } \mathcal{W}(b)(s) \texttt{ then}\hat{}(\bigsqcup_{i=0}^{\infty} \phi_i)(\tau) \texttt{ else } \{s\} \texttt{ fi} \sqsubseteq \bigsqcup_{i=0}^{\infty} \texttt{if } \mathcal{W}(b)(s) \texttt{ then } \hat{\phi}_i(\tau) \texttt{ else } \{s\} \texttt{ fi},$$

where $\tau = \mathcal{M}[\![P]\!](s)$. We distinguish three cases.

1. Let $\mathcal{W}(b)(s) = \perp_{W_\perp}$, then to prove: $\{\perp\} \sqsubseteq \bigsqcup_{i=0}^{\infty} \{\perp\}$, i.e., $\{\perp\} \sqsubseteq \{\perp\}$. This is true by Definition 6.31.

2. Let $\mathcal{W}(b)(s) = \mathit{ff}$, then to prove: $\{s\} \sqsubseteq \bigsqcup_{i=0}^{\infty} \{s\}$, i.e., $\{s\} \sqsubseteq \{s\}$. This is true by Definition 6.31.

3. Let $\mathcal{W}(b)(s) = \mathit{tt}$, then to prove: $\hat{}(\bigsqcup_{i=0}^{\infty} \phi_i)(\tau) \sqsubseteq \bigsqcup_{i=0}^{\infty} \hat{\phi}_i(\tau)$. If we can prove that $\forall \tau \in T : \hat{}(\bigsqcup_{i=0}^{\infty} \phi_i)(\tau) \sqsubseteq \bigsqcup_{i=0}^{\infty} \hat{\phi}_i(\tau)$, i.e., $\hat{}(\bigsqcup_{i=0}^{\infty} \phi_i) \sqsubseteq \bigsqcup_{i=0}^{\infty} \hat{\phi}_i$, we are finished. A proof of the continuity of $\hat{}$ is given in [de Bakker, 1980], from which we can conclude what needs to be proven.

□

Proof of Theorem 6.2: Immediate from Lemmas 6.2 and 6.3 and Fact 6.4. □

6.5 Equivalence of Operational and Denotational Semantics

In the previous section, we have given a denotational semantic function for meta-programs and we have proven that this function is well-defined. In Section 6.5.1, we will prove that the denotational semantics for meta-programs is equal to the operational semantics for meta-programs. From this we can conclude

that the denotational semantics of the interpreter of Section 6.3 is equal to the operational semantics of this interpreter. As the operational semantics of this interpreter is equal to the operational semantics of object-level 3APL, the denotational semantics of the interpreter is equal to the operational semantics of 3APL. One could thus argue that we give a denotational semantics for 3APL. This will be discussed in Section 6.5.2.

6.5.1 Equivalence Theorem

We prove the theorem using techniques from [Kuiper, 1981]. Kuiper proves equivalence of the operational and denotational semantics of a non-deterministic language with procedures but without a `while` construct. The proof involves structural induction on programs. As the cases of sequential composition and non-deterministic choice have been proven by Kuiper (and as they can easily be adapted to fit our language of meta-programs), we will only provide a proof for the atomic meta-actions and for the `while` construct. For a detailed explanation of the general ideas of the proof, we refer to [Kuiper, 1981].

In our proof, we will use a number of lemmas from Kuiper or slight variations thereof. We restate those results here, after which we present and prove the equivalence theorem.

Lemma 6.4 Let $\mathcal{W}(b)(s) = tt$. Then the following holds.

$$\begin{array}{lcl} \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) & = & \mathcal{O}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) \\ \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) & = & \mathcal{M}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) \end{array}$$

Lemma 6.5

$$\mathcal{O}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) = \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}) \circ \mathcal{O}(P')(s)$$

Lemma 6.6 For all $s \in S$ and $P \in Prog$ for which $\mathcal{C}(P)(s) \in \wp(S^+)$, $\mathcal{C}(P)(s)$ is a finite set.

Theorem 6.3 *($\mathcal{O}_m = \mathcal{M}$)* Let $\mathcal{O}_m : Prog \to (S_\perp \to \wp(S_\perp))$ be the operational semantics of meta-programs (Definition 6.18) and let $\mathcal{M} : Prog \to (S_\perp \to T)$ be the denotational semantics of meta-programs (Definition 6.32). Then, the following equivalence holds for all meta-programs $P \in Prog$ and all mental states $s \in S_\perp$.

$$\mathcal{O}_m(P)(s) = \mathcal{M}(P)(s)$$

Proof: The way to prove the equivalence result as was done by Kuiper, is the following. In case $\mathcal{C}_m(P)(s) \in \wp(S^+)$, induction on the *sum* of the lengths of the computation sequences in $\mathcal{C}_m(P)(s)$ is applied, thus proving $\mathcal{O}_m(P)(s) = \mathcal{M}(P)(s)$ in this case.[5] In case there is an infinite computation sequence in

[5] In the sequel, we will omit the subscript "m" to $\mathcal{C}$ and $\mathcal{O}$ which is used to denote that we are dealing with the meta-language.

$\mathcal{C}(P)(s)$ and so $\perp \in \mathcal{O}(P)(s)$, we prove $\mathcal{O}(P)(s) \setminus \{\perp\} \subseteq \mathcal{M}(P)(s)$ by induction on the length of *individual* computation sequences. This yields $\mathcal{O}(P) \sqsubseteq \mathcal{M}(P)$. Proving $\mathcal{M}(P) \sqsubseteq \mathcal{O}(P)$ by standard techniques then completes the proof.

$\mathcal{O}(P)(s) = \mathcal{M}(P)(s)$ holds trivially for $s = \perp$, so in the sequel we will assume $s \in S$.

1. $\mathcal{O}(P)(s) \sqsubseteq \mathcal{M}(P)(s)$

Case A: $\perp \notin \mathcal{O}(P)(s)$ i.e., $\mathcal{C}(P)(s) \in \wp(S^+)$

If $\mathcal{C}(P)(s) \in \wp(S^+)$, then we prove $\mathcal{O}(P)(s) = \mathcal{M}(P)(s)$ by cases, applying induction on the sum of the lengths of the computation sequences.

1. $P \equiv \mathit{execute}$
   Let $(\pi, \sigma) \in S$ and $\pi = a; \pi'$, with $\pi' \in \mathsf{Plan}$. If $\mathcal{T}(a, \sigma) = \sigma'$ (which implies that $a \in \mathsf{BasicAction}$), then the following can be derived directly from definitions 6.18, 6.16 and 6.32: $\mathcal{O}(\mathit{execute})(\pi, \sigma) = \kappa(\mathcal{C}(\mathit{execute})(\pi, \sigma)) = \{(\pi', \sigma')\} = \mathcal{M}(\mathit{execute})(\pi, \sigma)$. If $\mathcal{T}(a, \sigma)$ is undefined - meaning that either $a \in \mathsf{BasicAction}$ and $\mathcal{T}(a, \sigma)$ is undefined for this input, or $a \notin \mathsf{BasicAction}$ - we have $\mathcal{O}(\mathit{execute})(\pi, \sigma) = \emptyset = \mathcal{M}(\mathit{execute})(\pi, \sigma)$.

2. $P \equiv \mathit{apply}(\rho)$
   The proof is similar to the proof for *execute*.

3. $P \equiv \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}$
   In case $\mathcal{W}(b)(s) = \mathit{ff}$, we have that $\mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) = \{s\} = \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s)$ by definition. In the sequel, we will show that the equivalence also holds in case $\mathcal{W}(b)(s) = \mathit{tt}$.

   The function "length" yields the sum of the lengths of the computation sequences in a set. From the assumption that $\mathcal{C}(P)(s) \in \wp(S^+)$, we can conclude that $\mathcal{C}(P)(s)$ is a finite set (Lemma 6.6). From Definition 6.17, we can then conclude the following.

   $$\text{length}(\mathcal{C}(P')(s)) < \text{length}(\mathcal{C}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s)) < \infty$$

   $$\text{length}(\mathcal{C}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\kappa(\mathcal{C}(P')(s))) < \text{length}(\mathcal{C}(P'; \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s)) < \infty$$

   So, by induction we have:

   $$\begin{aligned} &\mathcal{O}(P')(s) && = \mathcal{M}(P')(s), \\ &\mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\kappa(\mathcal{C}(P')(s))) && = \mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\kappa(\mathcal{C}(P')(s))). \end{aligned}$$

The proof is then as follows.

$$
\begin{array}{ll}
\mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \\
= \mathcal{O}(P'; \texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(Lemma 6.4)} \\
= \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od}) \circ \mathcal{O}(P')(s) & \text{(Lemma 6.5)} \\
= \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od})(\kappa(\mathcal{C}(P')(s))) & \text{(Definition 6.18)} \\
= \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(\kappa(\mathcal{C}(P')(s))) & \text{(induction hypothesis)} \\
= \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(\mathcal{O}(P')(s)) & \text{(Definition 6.18)} \\
= \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od}) \circ \mathcal{M}(P')(s) & \text{(induction hypothesis)} \\
= \mathcal{M}(P'; \texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(Definition 6.32)} \\
= \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(Lemma 6.4)}
\end{array}
$$

Case B: $\bot \in \mathcal{O}(P)(s)$

If P and s are such that $\bot \in \mathcal{O}(P)(s)$ then we prove by cases that $\mathcal{O}(P)(s) \backslash \{\bot\} \subseteq \mathcal{M}(P)(s)$, applying induction on the length of the computation sequence corresponding to that outcome, i.e., we prove that for $s' \neq \bot$: $s' \in \mathcal{O}(P)(s) \Rightarrow s' \in \mathcal{M}(P)(s)$).

1. $P \equiv$ *execute* and $P \equiv$ *apply*(ρ)
   Equivalence was proven in case A.

2. $P \equiv \texttt{while } b \texttt{ do } P' \texttt{ od}$
   Consider a computation sequence $\delta = \langle s_1, \ldots, s_n (= s') \rangle \in \mathcal{C}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s)$. From Definition 6.17 of the function $\mathcal{C}$, we can conclude that there are intermediate states $s_j, s_{j+1} \neq \bot$ in this sequence δ, i.e., $\delta = \langle s_1, \ldots, s_j, s_{j+1}, \ldots, s_n (= s') \rangle$ (where s_1 can coincide with s_j), with $s_j = s_{j+1}$ and moreover: $\langle s_1, \ldots, s_j \rangle \in \mathcal{C}(P')(s)$ and $\langle s_{j+1}, \ldots, s_n \rangle \in \mathcal{C}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s_j)$. The following can be derived immediately from the above.

   $$
   \begin{array}{lcl}
   \text{length}(\langle s_1, \ldots, s_j \rangle) & < & \text{length}(\langle s_1, \ldots, s_n \rangle) \\
   \text{length}(\langle s_{j+1}, \ldots, s_n \rangle) & < & \text{length}(\langle s_1, \ldots, s_n \rangle)
   \end{array}
   $$

   We thus have the following induction hypothesis.

   $$
   \begin{array}{lcl}
   s_j \in \mathcal{O}(P')(s) & \Rightarrow & s_j \in \mathcal{M}(P')(s) \\
   s' \in \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s_j) & \Rightarrow & s' \in \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s_j)
   \end{array}
   $$

   As $\langle s_1, \ldots, s_j \rangle \in \mathcal{C}(P')(s)$, we know that $s_j \in \mathcal{O}(P')(s)$ (Definition 6.18) and similarly we can conclude that $s' \in \mathcal{O}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s_j)$. We thus have, using the induction hypothesis that: $s_j \in \mathcal{M}(P')(s)$ and $s' \in \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s_j)$. From this we can conclude the following, deriving what was to be proven.

   $$
   \begin{array}{lcll}
   s' & \in & \mathcal{M}(P'; \texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(Definition 6.32)} \\
   s' & \in & \mathcal{M}(\texttt{while } b \texttt{ do } P' \texttt{ od})(s) & \text{(Lemma 6.4)}
   \end{array}
   $$

2. $\mathcal{M}(P)(s) \sqsubseteq \mathcal{O}(P)(s)$

1. $P \equiv execute$
   Equivalence was proven in case (1).

2. $P \equiv apply(\rho)$
   Equivalence was proven in case (1).

3. $P \equiv \texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}$
   We will use induction on the entity $(i, length(P))$ where $length(P)$ denotes the length of the statement P and we use a lexicographic ordering on these entities, i.e., $(i_1, l_1) < (i_2, l_2)$ iff either $i_1 < i_2$ or $i_1 = i_2$ and $l_1 < l_2$. Clearly, $length(P') < length(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})$ holds. Therefore $(i, length(P')) < (i, length(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}))$ holds.

   We know that $\mathcal{M}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}) = \mu\Phi = \bigsqcup_{i=0}^{\infty} \Phi^i(\bot_{S_\bot \to T})$ by continuity of Φ (Theorem 6.2). Let $\phi_i = \Phi^i(\bot_{S_\bot \to T})$. We thus need to prove that $\bigsqcup_{i=0}^{\infty} \phi_i \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})$. So, if we can prove that $\phi_i \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})$ holds for all i, we will have the desired result. We will prove this by induction on the entity $(i, length(P))$. As $(i, length(P')) < (i, length(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}))$ and $(i, l) < (i + 1, l)$, our induction hypothesis will be:

   $$\mathcal{M}(P') \sqsubseteq \mathcal{O}(P') \text{ and } \phi_i \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od}).$$

   The induction basis is provided as $\phi_0 = \bot_{S_\bot \to T} \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})$ holds. From this we have to prove that for all $s \in S : \phi_{i+1}(s) \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s)$. Take an arbitrary $s \in S$. We have to prove that:

   $$\begin{array}{lcll} \Phi(\phi_i)(s) & \sqsubseteq & \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) & \text{i.e., by Definition 6.32} \\ \hat{\phi}_i(\mathcal{M}(P')(s)) & \sqsubseteq & \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s) & \text{i.e., by Lemma 6.4 and 6.5} \\ \hat{\phi}_i(\mathcal{M}(P')(s)) & \sqsubseteq & \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(\mathcal{O}(P')(s)) & (*). \end{array}$$

   We know that $\mathcal{M}(P')(s) = \mathcal{O}(P')(s)$ by the induction hypothesis and the fact that we have already proven $\mathcal{M}(P')(s) \sqsupseteq \mathcal{O}(P')(s)$. Let $\tau' = \mathcal{M}(P')(s) = \mathcal{O}(P')(s)$ and let $s' \in \tau'$. By the induction hypothesis, we have that $\phi_i(s') \sqsubseteq \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s')$ for all $s' \in S_\bot$. From this, we can conclude that $\bigcup_{s' \in \tau} \phi_i(s') \sqsubseteq \bigcup_{s' \in \tau} \mathcal{O}(\texttt{while}\ b\ \texttt{do}\ P'\ \texttt{od})(s')$ (see the proof of Lemma 6.2), which can be rewritten into what was to be proven $(*)$ using the definitions of $\hat{\phi}_i$ and function composition.

□

In Section 6.3, we stated that the object-level operational semantics of 3APL is equal to the meta-level operational semantics of the interpreter we specified in Definition 6.19. Above, we then stated that it holds for any meta-program that its operational semantics is equal to its denotational semantics. This holds in particular for the interpreter of Definition 6.19, i.e., we have the following corollary.

Corollary 6.1 *($\mathcal{O}_o = \mathcal{M}(\mathsf{interpreter})$)* From Theorems 6.1 and 6.3 we can conclude that the following holds.

$$\mathcal{O}_o = \mathcal{M}(\mathsf{interpreter})$$

6.5.2 Denotational Semantics of Object-Level 3APL

Corollary 6.1 states an equivalence between a denotational semantics and the object-level operational semantics for 3APL. The question is, whether this denotational semantics can be called a denotational semantics for object-level 3APL.

A denotational semantics for object-level 3APL should be a function taking a plan and a belief base and returning the result of executing the plan on this belief base, i.e., a function of type $\mathsf{Plan} \to (\Sigma_\perp \to \wp(\Sigma_\perp))$ or equivalently[6], of type $(\mathsf{Plan} \times \Sigma) \to \wp(\Sigma_\perp)$. The type of $\mathcal{M}(\mathsf{interpreter})$, i.e., $S_\perp \to \wp(S_\perp)$[7], does not match the desired type. This could however be remedied by defining the following function.

Definition 6.34 *($\mathcal{N}$)* Let *snd* be a function yielding the second element, i.e., the belief base, of a mental state in S and yielding $\perp_{\Sigma_\perp}$ for input $\perp$. This function is extended to handle sets of mental states through the function $\hat{}$, as was done in Definition 6.32. Then $\mathcal{N} : S_\perp \to \wp(\Sigma_\perp)$ is defined as follows.

$$\mathcal{N} = \lambda s \cdot \widehat{snd}(\mathcal{M}[\![\mathsf{interpreter}]\!](s))$$

Disregarding a $\perp$ input, the function $\mathcal{N}$ is of the desired type $(\mathsf{Plan} \times \Sigma) \to \wp(\Sigma_\perp)$. The question now is, whether it is legitimate to characterize the function $\mathcal{N}$ as being a denotational semantics for 3APL. The answer is no, because a denotational semantic function should be compositional in its program argument, which in this case is Plan. This is obviously *not* the case for the function $\mathcal{N}$ and therefore this function is not a denotational semantics for 3APL.

So, it seems that the specification of the denotational semantics for meta-programs cannot be used to define a denotational semantics for object-level 3APL. The difficulty of specifying a compositional semantic function is due to the nature of the plan revision rules: these rules can transform not just atomic statements, but any sequence of statements. The semantics of an atomic statement can thus depend on the statements around it. We will illustrate the problem using an example. In our example, we omit the belief condition from the plan revision rules for reasons of simplicity and presentation, and assume that the belief condition is true.

$$\begin{array}{lcl} a & \rightsquigarrow & b \\ b; c & \rightsquigarrow & d \\ c & \rightsquigarrow & e \end{array}$$

[6] For the sake of argument, we for the moment disregard a $\perp_{\Sigma_\perp}$ input.

[7] $\mathcal{M}(\mathsf{interpreter})$ is actually defined to be of type $S_\perp \to T$, but $T \subset \wp(S_\perp)$, so we may extend the result type to $\wp(S_\perp)$.

Now the question is, how we can define the semantics of $a; c$. Can it be defined in terms of the semantics of a and c? The semantics of a would have to be something involving the semantics of b and the semantics of c something with the semantics of e, taking into account the plan revision rules given above. The semantics of $a; c$ should however also be defined in terms of the semantics of d, because of the second plan revision rule: $a; c$ can be rewritten to $b; c$, which can be rewritten to d. Moreover, if b is not a basic action, the third rule cannot be applied and the semantics of e would be irrelevant. So, although we do not have a formal proof, it seems that the semantics of the sequential composition operator[8] of a 3APL plan or program cannot be defined using only the semantics of the parts of which the program is composed.

Another way to look at this issue is the following. In a regular procedural program, computation can be defined using the concept of a program counter. This counter indicates the location in the code of the statement that is to be executed next or the procedure that is to be called next. If a procedure is called, the program counter jumps to the body of this procedure. Computation of a 3APL program cannot be defined using such a counter. Consider for example the plan revision rules defined above and assume an initial plan $a; c$. Initially, the program counter would have to be at the start of this initial plan. Then the first plan revision rule is "called" and the counter jumps to b, i.e., the body of the first rule. According to the semantics of 3APL, it should be possible to get to the body of the second plan revision rule, as the statement being executed is $b; c$. There is however no reason for the program counter to jump from the body of the first rule to the body of the second rule.

6.6 Related Work and Conclusion

The concept of a meta-language for programming 3APL interpreters was first considered in [Hindriks et al., 1999a]. Our meta-language is similar to, but simpler than Hindriks' language. The main difference is that Hindriks includes constructs for explicit selection of a plan revision rule from a set of applicable ones. These constructs were not needed in this chapter. Dastani defines a meta-language for 3APL in [Dastani et al., 2003]. This language is similar to, but more extensive than Hindriks' language. Dastani's main contribution is the definition of constructs for explicit planning. Using these constructs, the possible outcomes of a certain sequence of rule applications and action executions can be calculated in advance, thereby providing the possibility to choose the most beneficial sequence. Contrary to our chapter, these papers do not discuss the relation between object-level and meta-level semantics, nor do they give a denotational semantics for the meta-language.

Concluding, we have proven equivalence of an operational and denotational semantics for a 3APL meta-language. We furthermore linked this 3APL meta-

[8] or actually of the plan concatenation operator •

language to object-level 3APL by proving equivalence between the semantics of a specific interpreter and object-level 3APL. Although these results were obtained for a simplified 3APL language, we conjecture that it will not be fundamentally more difficult to obtain similar results for full first order 3APL.[9]

As argued in the introduction, studying interpreter languages or meta-languages of agent programming languages is important. The research presented in this chapter provides an investigation into the semantics of such a meta-language, and shows how it is related to object-level 3APL. While it *is* possible to define a denotational semantics for the meta-language, it seems that it will be very difficult if not impossible to define a denotational semantics for object-level 3APL. Such a denotational semantics is important, especially if one is aiming for a compositional proof system for object-level 3APL.

Given that defining such a denotational semantics is problematic, a possible direction of research is to investigate whether it is possible to define a proof system for object-level 3APL that does not rely on a compositional semantics of plans. Researches along these lines will be presented in Chapter 7. Another direction of research, and one that will be followed in Chapter 8, is to restrict plan revision rules, such that the semantics of plans becomes compositional.

[9]The requirement of bounded non-determinism will in particular not be violated.

Chapter 7

Dynamic Logic for Plan Revision

This chapter is based on [van Riemsdijk et al., 2005d] and [van Riemsdijk et al., 2005e]. The agent programming language we consider in this chapter is essentially the object-level 3APL language of Chapter 6, with some minor modifications. We investigate the possibility of defining a logic for reasoning about this 3APL language. This logic will have to be designed such that it can somehow handle the non-compositional nature of the semantics of 3APL plans, which arises if plans can be revised using plan revision rules. The logic that we present is a dynamic logic.

The outline of this chapter is as follows. In Section 7.1, we address related work. After defining 3APL and its semantics (Section 7.2), we propose a dynamic logic for proving properties of 3APL plans in the context of plan revision rules (Section 7.3). As will become clear, this is actually not a logic for general 3APL plans, but the plans that the logic can deal with are restricted in a certain way. For this logic, we provide a sound and complete axiomatization (Section 7.4). In Section 7.5, we discuss how this logic for restricted 3APL plans can be extended to a logic for non-restricted plans and we discuss some example proofs, using the logic. Finally, we consider the relation between proving properties of procedural programs and proving properties of 3APL agents in Section 7.6. In particular, we compare procedures with plan revision rules.

To the best of our knowledge, this is the first attempt to design a logic and deductive system for plan revision rules or similar language constructs. Considering the semantic difficulties that arise with the introduction of this type of construct, it is not a priori obvious that it would be possible at all to design a deductive system to reason about these constructs. The main aim of this work was thus to investigate whether it is possible to define such a system, and in this way also to get a better theoretical understanding of the construct of plan revision rules. Whether the system presented in this chapter is also practically useful to verify 3APL agents, remains to be seen and will be subject to further research.

7.1 Related Work

This research builds on a body of work done in the area of theoretical computer science on *formal semantics* and *logics* of programming languages (see, e.g., [de Bakker, 1980]). A formal semantics for a programming language is used to formally specify the meaning of the programs written in this language. Specifying the meaning of a programming language using formal semantics is important for a number of reasons. For example, the specification of a formal semantics can be used to identify issues and problems with the language. Defining the semantics forces one to be precise, which might uncover problems which were overlooked before. Also, the semantics can serve as a basis for comparing various languages. Further, and most important in this context, it is a necessary prerequisite if one wants to do formal verification of programs written in some language. One cannot claim to have proven that a program satisfies a certain property, without knowing exactly what this program does.

Semantics of programming languages can be defined in different ways. One kind of semantics is the operational semantics, which has been used several times in this thesis. The operational semantics of plan revision rules, which is important in this chapter, is similar to that of procedures in procedural programming. In fact, plan revision rules can be viewed as an extension of procedures. Logics and semantics for procedural languages are for example studied in [de Bakker, 1980]. Although the operational semantics of procedures and plan revision rules are similar, techniques for reasoning about procedures cannot be used for plan revision rules. This is due to the fact that the introduction of these rules results in the semantics of the sequential composition operator no longer being compositional. We elaborate on this issue in Sections 7.3 and 7.6.

With respect to verification, there are in general two approaches: model checking [E.M.Clarke et al., 2000] and theorem proving (see, e.g., [de Bakker, 1980]). In model checking, a model is built describing the execution of the program, and it is checked whether some temporal property is satisfied by this model. An example of work on model checking in the area of agent programming languages is [Bordini et al., 2003], in which model checking of the agent programming language AgentSpeak is addressed.

In theorem proving, on which we focus in this chapter, a program is proven to satisfy a certain property using a logic with deductive system or axiomatization. Various logics can be used for this purpose, such as Hoare logic (see, e.g., [Apt, 1981] for a survey) and dynamic logic [Harel et al., 2000], which we use in this chapter. In the context of 3APL, a sketch of a dynamic logic to reason about programs written in this language has been given in [van Riemsdijk et al., 2003b]. This logic however is designed to reason about a 3APL interpreter language or meta-language, whereas in this chapter we take a different viewpoint and reason about plans. In [Hindriks et al., 2000], a programming logic (without axiomatization) was given for a fragment of 3APL without plan revision rules.

Further, we mention related work in the field of *planning*. In general, plan-

ning deals with the problem of how to get from some current state to a desired goal state through a sequence of actions forming the plan. The way this problem is approached in this field, is by *searching* for an appropriate plan using a specification of the available actions and their preconditions and effects [Fikes and Nilsson, 1971]. The search space can however become quite large in realistic problems. Part of the planning research thus involves the investigation of more efficient ways in which this search can be performed, and the development of heuristics to guide this search.

While the general objective of planning, i.e., generating a plan with which the agent can achieve its goals, is closely related to the objective of cognitive agent programming, the techniques that are used in the two fields differ. Planning involves *reasoning* about the effects of actions, while in a programming context it is the *programmer* who defines the available plans, together with the situations in which they might be executed. In fact, the so-called Procedural Reasoning System [Georgeff and Lansky, 1987] on which most of today's agent programming languages, including 3APL, are directly or indirectly based, was proposed as an alternative to the traditional planning systems.[1] It was in part motivated by the observation that those systems require search through potentially large search spaces. In contrast with research in the field of planning, research regarding agent programming languages involves the design and investigation of appropriate *programming constructs* for the specification of plans, and in this chapter we are concerned with the programming construct of plan revision rules of the 3APL language in particular. Also, the structure of plans generally differs between the two fields: in planning, a plan often consists of a partially ordered set of actions, while in agent programming the structure of plans is generally simpler.

Nevertheless, the general idea of plan revision as incorporated in 3APL is also being investigated in the field of planning. In that context, it is mostly referred to as *plan repair*. The motivation for that work is similar to the motivation of the addition of plan revision capabilities to 3APL agents, i.e., things may go wrong during execution of a plan since the agent executes its plans in some environment, and in that case the agent will have to replan, or to adapt the old plan to the changed circumstances (see, e.g., [Hammond, 1990, van der Krogt and de Weerdt, 2005a]). In theory, modifying an existing plan is (worst-case) no more efficient than a complete replanning [Nebel and Koehler, 1995], but the idea is that in practice, plan repair is often more efficient [van der Krogt and de Weerdt, 2005a].

While approaches to plan repair mostly involve reasoning about actions, there are also some approaches which use precompiled plans to do plan repair, such as [van der Krogt and de Weerdt, 2005b, Drabble et al., 1997]. The latter

[1] An exception to this distinction between planning and programming is formed by the language ConGolog [Giacomo et al., 2000], in which both approaches are combined. ConGolog is based on the situation calculus, and a program written in the language is used to constrain the search space for finding an appropriate plan.

approaches are somewhat more closely related to plan revision as done in 3APL, since the way in which plans may be repaired is prespecified in both approaches. Nevertheless, these approaches to plan repair are embedded in a general planning framework, and the exact relation with plan revision as used in 3APL is not immediately clear. Investigations along these lines fall outside the scope of this chapter, but form an interesting issue for future research.

7.2 3APL

7.2.1 Syntax

The languages of belief bases and plans are the same as in Chapter 6 (Definitions 6.1 and 6.2). Note, however, that in this chapter we use p as typical element of the language $\mathcal{L}$.

Definition 7.1 *(belief base)* Assume a propositional language $\mathcal{L}$ with typical formula p and the connectives $\wedge$ and $\neg$ with the usual meaning. Then the set of belief bases Σ with typical element σ is defined to be $\wp(\mathcal{L})$.

Definition 7.2 *(plan)* Assume that a set $\mathsf{BasicAction}$ with typical element a is given, together with a set $\mathsf{AbstractPlan}$ with typical element p.[2] Let $c \in (\mathsf{BasicAction} \cup \mathsf{AbstractPlan})$. Then the set of plans Plan with typical element π is defined as follows.

$$\pi ::= a \mid p \mid c; \pi$$

We use ϵ to denote the empty plan. The concatenation of a plan π and the empty plan is equal to π, i.e., $\epsilon; \pi$ and $\pi; \epsilon$ are taken to be identical to π (see Chapter 2 (Definition 2.8 and Remark 2.1) for a rigorous explanation).

Plan revision rules are as in Chapters 2 and 6, except that we omit the belief condition from the rules here. The approach presented in this chapter could be extended to rules with belief condition without fundamental difficulties. We omit the belief condition from plan revision rules since we want to focus on the plan revision aspect of these rules.

Definition 7.3 *(plan revision rules)* The set of plan revision rules $\mathcal{R}$ is defined as follows: $\mathcal{R} = \{\pi_h \rightsquigarrow \pi_b \mid \pi_h, \pi_b \in \mathsf{Plan}\}$.

Note that π_h may not be the empty plan, since ϵ is not an element of the language of plans Plan. Below, we provide the definition of a 3APL agent. It will become clear in the sequel that for the investigations carried out in this chapter, it is not necessary to endow the agent with an initial belief base and plan.

[2]Note that we use p to denote an element from the propositional language $\mathcal{L}$, as well as an element from $\mathsf{AbstractPlan}$. It will however be indicated explicitly which kind of element is meant.

Definition 7.4 *(3APL agent)* A 3APL agent $\mathcal{A}$ is a tuple $\langle \mathsf{Rule}, \mathcal{T} \rangle$ where $\mathsf{Rule} \subseteq \mathcal{R}$ is a finite set of plan revision rules and $\mathcal{T} : (\mathsf{BasicAction} \times \Sigma) \to \Sigma$ is a partial function, expressing how belief bases are updated through basic action execution.

Configurations are defined as in Chapter 6 (Definition 6.5), except that they were called "mental states" in that chapter.

Definition 7.5 *(configuration)* Let Σ be the set of belief bases and let Plan be the set of plans. Then $\mathsf{Plan} \times \Sigma$ is the set of configurations of a 3APL agent.

7.2.2 Semantics

The semantics of a programming language can be defined as a function taking a statement and a state, and yielding the set of states resulting from executing the initial statement in the initial state. In this way, a statement can be viewed as a transformation function on states. In (object-level) 3APL, plans can be seen as statements and belief bases can be seen as states on which these plans operate. The kind of semantics we are concerned with in this chapter, is an operational semantics.

As usual in this thesis, we define the operational semantics of 3APL using a transition system [Plotkin, 1981]. This transition system $\mathsf{Trans}_{\mathcal{A}}$ is specific to 3APL agent $\mathcal{A}$. We take $\mathcal{A}$ to be $\langle \mathsf{Rule}, \mathcal{T} \rangle$, and we assume $\mathcal{A}$ to have a set of basic actions $\mathsf{BasicAction}$.

As in Chapter 6, there are two kinds of transitions, i.e., transitions describing the execution of basic actions and those describing the application of a plan revision rule. The transitions are labeled to denote the kind of transition (although the labeling differs slightly from the labeling in Chapter 6). The transition rule for application of a plan revision rule differs from the plan revision transition rule of Chapter 6 (Definition 6.9), since we omit the belief condition from plan revision rules in this chapter.

Definition 7.6 *(action execution)* Let $a \in \mathsf{BasicAction}$.

$$\frac{\mathcal{T}(a, \sigma) = \sigma'}{\langle a; \pi, \sigma \rangle \rightarrow_{exec} \langle \pi, \sigma' \rangle}$$

Definition 7.7 *(rule application)* Let $\rho : \pi_h \rightsquigarrow \pi_b \in \mathsf{Rule}$.

$$\langle \pi_h \bullet \pi, \sigma \rangle \rightarrow_{app} \langle \pi_b \bullet \pi, \sigma \rangle$$

In the sequel, it will be useful to have a function taking a plan revision rule and a plan, and yielding the plan resulting from the application of the rule to this given plan. Based on this function, we also define a function taking a set of plan revision rules and a plan and yielding the set of rules applicable to this plan.

Definition 7.8 *(rule application)* Let $\mathcal{R}$ be the set of plan revision rules and let Plan be the set of plans. Let $\rho : \pi_h \rightsquigarrow \pi_b \in \mathcal{R}$ and $\pi, \pi' \in \mathsf{Plan}$. The partial function $apply : (\mathcal{R} \times \mathsf{Plan}) \rightarrow \mathsf{Plan}$ is then defined as follows.

$$apply(\rho)(\pi) = \begin{cases} \pi_b \bullet \pi' & \text{if } \pi = \pi_h \bullet \pi', \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The function $applicable : (\wp(\mathcal{R}) \times \mathsf{Plan}) \rightarrow \wp(\mathcal{R})$ yielding the set of rules applicable to a certain plan, is then as follows: $applicable(\mathsf{Rule}, \pi) = \{\rho \in \mathsf{Rule} \mid apply(\rho)(\pi) \text{ is defined}\}$.

As in Chapter 6 (Definition 6.16), the operational semantics is defined using the notion of computation sequences. The function for calculating computation sequences is parameterized by 3APL agent $\mathcal{A}$. In this chapter, we need to make the parametrization with agent $\mathcal{A}$ explicit in order to define the dynamic logic properly.

Definition 7.9 *(computation sequences)* The set Σ^+ of finite computation sequences is defined as $\{\sigma_1, \ldots, \sigma_i, \ldots, \sigma_n \mid \sigma_i \in \Sigma, 1 \leq i \leq n, n \in \mathbb{N}\}$.

Definition 7.10 *(function for calculating computation sequences)* Let $x_i \in \{exec, app\}$ for $1 \leq i \leq m$. The function $\mathcal{C}^{\mathcal{A}} : (\mathsf{Plan} \times \Sigma) \rightarrow \wp(\Sigma^+)$ is then as defined below.

$$\mathcal{C}^{\mathcal{A}}(\pi, \sigma) = \{\sigma, \ldots, \sigma_m \in \Sigma^+ \mid \langle \pi, \sigma \rangle \rightarrow_{x_1} \ldots \rightarrow_{x_m} \langle \epsilon, \sigma_m \rangle \text{ is a finite sequence of transitions in } \mathsf{Trans}_{\mathcal{A}}\}.$$

Note that we only take into account successfully terminating transition sequences, i.e., those sequences ending in a configuration with an empty plan. Explicitly taking into account non-terminating transition sequences in the semantics as was done in Chapter 6 (Definition 6.17), is not needed in this chapter. The semantics of the dynamic logic is defined on the basis of the operational semantics that will be introduced below, which is defined on the basis of successfully terminating transition sequences.

Definition 7.11 *(operational semantics)* Let $\kappa : \Sigma^+ \rightarrow \Sigma$ be a function yielding the last element of a finite computation sequence, extended to handle sets of computation sequences as follows, where I is some set of indices: $\kappa(\{\delta_i \mid i \in I\}) = \{\kappa(\delta_i) \mid i \in I\}$. The operational semantic function $\mathcal{O}^{\mathcal{A}} : \mathsf{Plan} \rightarrow (\Sigma \rightarrow \wp(\Sigma))$ is defined as follows:

$$\mathcal{O}^{\mathcal{A}}(\pi)(\sigma) = \kappa(\mathcal{C}^{\mathcal{A}}(\pi, \sigma)).$$

We will sometimes omit the superscript $\mathcal{A}$ to functions as defined above, for reasons of presentation.

Example 7.1 Let $\mathcal{A}$ be an agent with plan revision rules $\{p; a \rightsquigarrow b, p \rightsquigarrow c\}$, where p is an abstract plan and a, b, c are basic actions. Let σ_a be the belief base resulting from the execution of a in σ, i.e., $\mathcal{T}(a, \sigma) = \sigma_a$, let σ_{ab} be the belief base resulting from executing first a and then b in σ, etc.

Then $\mathcal{C}^{\mathcal{A}}(p; a)(\sigma) = \{(\sigma, \sigma, \sigma_b), (\sigma, \sigma, \sigma_c, \sigma_{ca})\}$, which is based on the transition sequences $\langle p; a, \sigma\rangle \rightarrow_{app} \langle b, \sigma\rangle \rightarrow_{exec} \langle \epsilon, \sigma_b\rangle$ and $\langle p; a, \sigma\rangle \rightarrow_{app} \langle c; a, \sigma\rangle \rightarrow_{exec} \langle a, \sigma_c\rangle \rightarrow_{exec} \langle \epsilon, \sigma_{ca}\rangle$. We thus have that $\mathcal{O}^{\mathcal{A}}(p; a)(\sigma) = \{\sigma_b, \sigma_{ca}\}$. △

7.3 Plan Revision Dynamic Logic

In programming language research, an important area is the specification and verification of programs. Program logics are designed to facilitate this process. One such logic is dynamic logic [Harel, 1979, Harel et al., 2000], which we are concerned with in this chapter. In dynamic logic, programs are explicit syntactic constructs in the logic. To be able to discuss the effect of the execution of a program π on the truth of a formula ϕ, the modal construct $[\pi]\phi$ is used. This construct intuitively states that in all states in which π halts, the formula ϕ holds.

Programs in general are constructed from atomic programs and composition operators. An example of a composition operator is the sequential composition operator (;), where the program $\pi_1; \pi_2$ intuitively means that π_1 is executed first, followed by the execution of π_2. The semantics of such a compound program can in general be determined by the semantics of the parts of which it is composed. This compositionality property allows analysis by structural induction (see also [van Emde Boas, 1978]), i.e., analysis of a compound statement by analysis of its parts. Analysis of the sequential composition operator by structural induction can in dynamic logic be expressed by the following formula, which is usually a validity: $[\pi_1; \pi_2]\phi \leftrightarrow [\pi_1][\pi_2]\phi$. For 3APL plans on the contrary, this formula does not always hold. This is due to the presence of plan revision rules.

We will informally explain this using the 3APL agent of Example 7.1. As explained, the operational semantics of this agent, given initial plan $p; a$ and initial state σ, is as follows: $\mathcal{O}(p; a)(\sigma) = \{\sigma_b, \sigma_{ca}\}$. Now compare the result of first "executing"[3] p in σ and then executing a in the resulting belief base, i.e., compare the set $\mathcal{O}(a)(\mathcal{O}(p)(\sigma))$. In this case, there is only one successfully terminating transition sequence and it ends in σ_{ca}, i.e., $\mathcal{O}(a)(\mathcal{O}(p)(\sigma)) = \{\sigma_{ca}\}$. Now, if it would be the case that $\sigma_{ca} \models \phi$ but $\sigma_b \not\models \phi$, the formula $[p; a]\phi \leftrightarrow [p][a]\phi$ would not hold. In particular, the implication would not hold from right to left.

[3] We will use the word "execution" in two ways. Firstly, as in this context, we will use it to denote the execution of an arbitrary plan in the sense of going through several transition of type *exec* or *app*, starting in a configuration with this plan and resulting in some final configurations. Secondly, we will use it to refer to the execution of a basic action in the sense of going through a transition of type *exec*.

Analysis of plans by structural induction in this way thus does not work for 3APL. In order to be able to prove correctness properties of 3APL programs however, one can perhaps imagine that it is important to have *some* kind of induction. As we will show in the sequel, the kind of induction that can be used to reason about 3APL programs, is induction on the *number of plan revision rule applications in a transition sequence*. We will introduce a dynamic logic for 3APL based on this idea.

7.3.1 Syntax

In order to be able to do induction on the number of plan revision rule applications in a transition sequence, we introduce so-called *restricted plans*. These are plans, annotated with a natural number[4]. Informally, if the restriction parameter of a plan is n, the number of rule applications during execution of this plan cannot exceed n.

Definition 7.12 *(restricted plans)* Let Plan be the language of plans and let $\mathbb{N}^- = \mathbb{N} \cup \{-1\}$. Then, the language Plan_r of restricted plans is defined as $\{\pi\restriction_n \mid \pi \in \mathsf{Plan}, n \in \mathbb{N}^-\}$.

Below, we define the language of dynamic logic in which properties of 3APL agents can be expressed. In the logic, one can express properties of restricted plans. As will become clear in the sequel, one can prove properties of the plan of a 3APL agent by proving properties of restricted plans.

Definition 7.13 *(plan revision dynamic logic (*PRDL*))* Let $\pi\restriction_n \in \mathsf{Plan}_r$ be a restricted plan. Then the language of dynamic logic $\mathcal{L}_{\mathsf{PRDL}}$ with typical element ϕ is defined as follows:

- $\mathcal{L} \subseteq \mathcal{L}_{\mathsf{PRDL}}$,
- if $\phi \in \mathcal{L}_{\mathsf{PRDL}}$, then $[\pi\restriction_n]\phi \in \mathcal{L}_{\mathsf{PRDL}}$,
- if $\phi, \phi' \in \mathcal{L}_{\mathsf{PRDL}}$, then $\neg\phi \in \mathcal{L}_{\mathsf{PRDL}}$ and $\phi \wedge \phi' \in \mathcal{L}_{\mathsf{PRDL}}$.

7.3.2 Semantics

In order to define the semantics of PRDL, we first define the semantics of restricted plans. As for ordinary plans, we also define an operational semantics for restricted plans. We do this by defining a function for calculating computation sequences, given an initial restricted plan and a belief base.

Definition 7.14 *(function for calculating computation sequences)* Let $x_i \in \{exec, app\}$ for $1 \leq i \leq m$. Let $N_{app}(\theta)$ be a function yielding the number

[4]Or with the number -1. It will become clear in the sequel why we need this.

of transitions of the form $s_i \rightarrow_{app} s_{i+1}$ in the sequence of transitions θ. The function $\mathcal{C}_r^{\mathcal{A}} : (\mathsf{Plan}_r \times \Sigma) \rightarrow \wp(\Sigma^+)$ is then as defined below.

$$\mathcal{C}_r^{\mathcal{A}}(\pi\restriction_n, \sigma) = \{\sigma, \ldots, \sigma_m \in \Sigma^+ \mid \theta = \langle \pi, \sigma\rangle \rightarrow_{x_1} \ldots \rightarrow_{x_m} \langle \epsilon, \sigma_m\rangle \text{ is a finite sequence of transitions in } \mathsf{Trans}_{\mathcal{A}} \text{ where } 0 \leq N_{app}(\theta) \leq n\}$$

As one can see in the definition above, the computation sequences $\mathcal{C}_r^{\mathcal{A}}(\pi\restriction_n, \sigma)$ are based on transition sequences starting in configuration $\langle \pi, \sigma\rangle$. The number of rule applications in these transition sequences should be between 0 and n, in contrast with the function $\mathcal{C}^{\mathcal{A}}$ of Definition 7.10, in which there is no restriction on this number.

Based on the function $\mathcal{C}_r^{\mathcal{A}}$, we define the operational semantics of restricted plans by taking the last elements of the computation sequences yielded by $\mathcal{C}_r^{\mathcal{A}}$. The set of belief bases is empty if the restriction parameter is equal to -1.

Definition 7.15 *(operational semantics)* Let κ be as in Definition 7.11. The operational semantic function $\mathcal{O}_r^{\mathcal{A}} : \mathsf{Plan}_r \rightarrow (\Sigma \rightarrow \wp(\Sigma))$ is defined as follows:

$$\mathcal{O}_r^{\mathcal{A}}(\pi\restriction_n)(\sigma) = \begin{cases} \kappa(\mathcal{C}_r^{\mathcal{A}}(\pi\restriction_n, \sigma)) & \text{if } n \geq 0, \\ \emptyset & \text{if } n = -1. \end{cases}$$

Using the operational semantics of restricted plans, we can now define the semantics of plan revision dynamic logic.

Definition 7.16 *(semantics of* PRDL*)* Let $p \in \mathcal{L}$ be a propositional formula, let $\phi, \phi' \in \mathcal{L}_{\mathsf{PRDL}}$ and let $\models_{\mathcal{L}}$ be the entailment relation defined for $\mathcal{L}$ as usual. The semantics $\models_{\mathcal{A}}$ of $\mathcal{L}_{\mathsf{PRDL}}$ is then as defined below.

$$\begin{array}{lcl} \sigma \models_{\mathcal{A}} p & \Leftrightarrow & \sigma \models_{\mathcal{L}} p \\ \sigma \models_{\mathcal{A}} [\pi\restriction_n]\phi & \Leftrightarrow & \forall \sigma' \in \mathcal{O}_r^{\mathcal{A}}(\pi\restriction_n)(\sigma) : \sigma' \models_{\mathcal{A}} \phi \\ \sigma \models_{\mathcal{A}} \neg\phi & \Leftrightarrow & \sigma \not\models_{\mathcal{A}} \phi \\ \sigma \models_{\mathcal{A}} \phi \wedge \phi' & \Leftrightarrow & \sigma \models_{\mathcal{A}} \phi \text{ and } \sigma \models_{\mathcal{A}} \phi' \end{array}$$

We use the subscript $\mathcal{A}$ to the satisfaction relation for PRDL, to indicate that the semantics is relative to agent $\mathcal{A}$. Let $\mathsf{Rule} \subseteq \mathcal{R}$ be a finite set of plan revision rules. Then, if $\forall \mathcal{T}, \sigma : \sigma \models_{\langle \mathsf{Rule}, \mathcal{T}\rangle} \phi$, we write $\models_{\mathsf{Rule}} \phi$.

7.4 The Axiom System

In order to prove properties of restricted plans, we propose a deductive system for PRDL in this section. Rather than proving properties of restricted plans, the aim is however to prove properties of non-restricted 3APL plans. The idea is that this can be done using the axiom system for restricted plans, by relating the semantics of restricted plans to that of non-restricted plans. We will explain and elaborate on this in Section 7.5.

Definition 7.17 *(axiom system ($\mathsf{AS}_{\mathsf{Rule}}$))* Let BasicAction be a set of basic actions, AbstractPlan be a set of abstract plans and Rule $\subseteq \mathcal{R}$ be a finite set of plan revision rules. Let $a \in$ BasicAction, let $p \in$ AbstractPlan, let $c \in$ (BasicAction $\cup$ AbstractPlan) and let ρ range over $applicable(\mathsf{Rule}, c; \pi)$. The following are then the axioms of the system $\mathsf{AS}_{\mathsf{Rule}}$.

(PRDL1)	$[\pi\restriction_{-1}]\phi$	
(PRDL2)	$[p\restriction_0]\phi$	
(PRDL3)	$[\epsilon\restriction_n]\phi \leftrightarrow \phi$	with $0 \leq n$
(PRDL4)	$[c;\pi\restriction_n]\phi \leftrightarrow [c\restriction_0][\pi\restriction_n]\phi \wedge \bigwedge_\rho [apply(\rho, c; \pi)\restriction_{n-1}]\phi$	with $0 \leq n$
(PL)	axioms for propositional logic	
(PDL)	$[\pi\restriction_n](\phi \rightarrow \phi') \rightarrow ([\pi\restriction_n]\phi \rightarrow [\pi\restriction_n]\phi')$	

The following are the rules of the system $\mathsf{AS}_{\mathsf{Rule}}$.

(GEN)
$$\frac{\phi}{[\pi\restriction_n]\phi}$$

(MP)
$$\frac{\phi_1,\ \phi_1 \rightarrow \phi_2}{\phi_2}$$

As the axiom system is relative to a given set of plan revision rules Rule, we will use the notation $\vdash_{\mathsf{Rule}} \phi$ to specify that ϕ is derivable in the system $\mathsf{AS}_{\mathsf{Rule}}$ above.

We will now explain the PRDL axioms of the system. The other axioms and the rules are standard for propositional dynamic logic (PDL) [Harel, 1979]. We start by explaining the most interesting axiom: (PRDL4). We first observe that there are two types of transitions that can be derived for a 3APL agent: action execution and rule application (see Definitions 7.6 and 7.7). Consider a configuration $\langle a; \pi, \sigma\rangle$ where a is a basic action. Then during computation, possible next configurations are $\langle \pi, \sigma'\rangle$[5] (action execution) and $\langle apply(\rho, a; \pi), \sigma\rangle$ (rule application) where ρ ranges over the applicable rules, i.e., $applicable(\mathsf{Rule}, a; \pi)$.[6] We can thus analyze the plan $a; \pi$ by analyzing π after the execution of a, and the plans resulting from applying a rule, i.e., $apply(\rho, a; \pi)$.[7] The execution of an action can be represented by the number 0 as restriction parameter, yielding the first term of the right-hand side of (PRDL4): $[a\restriction_0][\pi\restriction_n]\phi$.[8] The second term is a conjunction of $[apply(\rho, c; \pi)\restriction_{n-1}]\phi$ over all applicable rules ρ. The

[5] assuming that $\mathcal{T}(a, \sigma) = \sigma'$

[6] See Definition 7.8 for the definitions of the functions *apply* and *applicable*.

[7] Note that one could say we analyze a plan $a; \pi$ partly by structural induction, as it is partly analyzed in terms of a and π.

[8] In our explanation, we consider the case where c is a basic action, but the axiom holds also for abstract plans.

restriction parameter is $n-1$ as we have "used" one of our n permitted rule applications. The first three axioms represent basic properties of restricted plans. (PRDL1) can be used to eliminate the second term on the right-hand side of axiom (PRDL4), if the left-hand side is $[c;\pi\restriction_0]\phi$. (PRDL2) can be used to eliminate the first term on the right-hand side of (PRDL4), if c is an abstract plan. As abstract plans can only be transformed through rule application, there will be no resulting states if the restriction parameter of the abstract plan is 0, i.e., if no rule applications are allowed. (PRDL3) states that if ϕ is to hold after execution of the empty plan, it should hold "now". It can be used to derive properties of an atomic plan c, by using axiom (PRDL4) with the plan $c;\epsilon$.

7.4.1 Soundness

The axiom system of Definition 7.17 is sound.

Theorem 7.1 *(soundness)* Let $\phi \in \mathcal{L}_{\mathsf{PRDL}}$. Let $\mathsf{Rule} \subseteq \mathcal{R}$ be an arbitrary finite set of plan revision rules. Then the axiom system $\mathsf{AS}_{\mathsf{Rule}}$ is sound, i.e.:

$$\vdash_{\mathsf{Rule}} \phi \;\Rightarrow \models_{\mathsf{Rule}} \phi.$$

Proof: We prove soundness of the PRDL axioms of the system $\mathsf{AS}_{\mathsf{Rule}}$. In the following, let $\pi \in \mathsf{Plan}$ be an arbitrary plan and let $\phi \in \mathcal{L}_{\mathsf{PRDL}}$ be an arbitrary PRDL formula. Furthermore, $\mathcal{A} = \langle \mathsf{Rule}, \mathcal{T} \rangle$ and $\models_{\langle \mathsf{Rule}, \mathcal{T} \rangle}$ will be abbreviated by $\models_{\mathsf{Rule}}$.

(PRDL1) To prove: $\forall \mathcal{T}, \sigma : \sigma \models_{\mathsf{Rule}} [\pi\restriction_{-1}]\phi$. Let $\sigma \in \Sigma$ be an arbitrary belief base and let $\mathcal{T}$ be an arbitrary belief update function. We have that $\sigma \models_{\mathsf{Rule}} [\pi\restriction_{-1}]\phi \Leftrightarrow \forall \sigma' \in \mathcal{O}_r^{\mathcal{A}}(\pi\restriction_{-1})(\sigma) : \sigma' \models_{\mathsf{Rule}} \phi$ by Definition 7.16. Furthermore, $\mathcal{O}_r^{\mathcal{A}}(\pi\restriction_{-1})(\sigma) = \emptyset$ by Definition 7.15, trivially yielding the desired result.

(PRDL2) Let $p \in \mathsf{AbstractPlan}$ be an arbitrary abstract plan. To prove: $\forall \mathcal{T}, \sigma : \sigma \models_{\mathsf{Rule}} [p\restriction_0]\phi$. Let $\sigma \in \Sigma$ be an arbitrary belief base and let $\mathcal{T}$ be an arbitrary belief update function. We have that $\sigma \models_{\mathsf{Rule}} [p\restriction_0]\phi \Leftrightarrow \forall \sigma' \in \mathcal{O}_r^{\mathcal{A}}(p\restriction_0)(\sigma) : \sigma' \models_{\mathsf{Rule}} \phi$ by Definition 7.16. Furthermore, $\mathcal{O}_r^{\mathcal{A}}(p\restriction_0)(\sigma) = \emptyset$ by Definitions 7.15, 7.14, 7.6, and 7.7, trivially yielding the desired result.

(PRDL3) To prove: $\forall \mathcal{T}, \sigma : \sigma \models_{\mathsf{Rule}} [\epsilon\restriction_n]\phi \leftrightarrow \phi$ where $n \geq 0$, i.e., $\forall \mathcal{T}, \sigma : (\sigma \models_{\mathsf{Rule}} [\epsilon\restriction_n]\phi \Leftrightarrow \sigma \models_{\mathsf{Rule}} \phi)$. Let $\sigma \in \Sigma$ be an arbitrary belief base and let $\mathcal{T}$ be an arbitrary belief update function. By Definition 7.14, we have that $\mathcal{C}_r^{\mathcal{A}}(\epsilon\restriction_n, \sigma) = \{\sigma\}$ where $n \geq 0$, i.e.:

$$\kappa(\mathcal{C}_r^{\mathcal{A}}(\epsilon\restriction_n, \sigma)) = \{\sigma\}. \tag{7.1}$$

By Definitions 7.16, 7.15 and (7.1), we have the following, yielding the desired result.

$$\begin{array}{rcl} \sigma \models_{\mathsf{Rule}} [\epsilon\restriction_n]\phi & \Leftrightarrow & \forall \sigma' \in \mathcal{O}_r^{\mathcal{A}}(\epsilon\restriction_n)(\sigma) : \sigma' \models_{\mathsf{Rule}} \phi \\ & \Leftrightarrow & \forall \sigma' \in \kappa(\mathcal{C}_r^{\mathcal{A}}(\epsilon\restriction_n, \sigma)) : \sigma' \models_{\mathsf{Rule}} \phi \\ & \Leftrightarrow & \sigma \models_{\mathsf{Rule}} \phi \end{array}$$

(PRDL4) To prove: $\forall \mathcal{T}, \sigma : \sigma \models_{\langle \mathsf{Rule}, \mathcal{T} \rangle} [c; \pi\restriction_n]\phi \leftrightarrow [c\restriction_0][\pi\restriction_n]\phi \wedge \bigwedge_\rho [apply(\rho, c; \pi)\restriction_{n-1}]\phi$. Let $\sigma \in \Sigma$ be an arbitrary belief base and let $\mathcal{T}$ be an arbitrary belief update function. Then to prove:

$$\sigma \models_{\langle \mathsf{Rule}, \mathcal{T} \rangle} [c; \pi\restriction_n]\phi \Leftrightarrow$$
$$\sigma \models_{\langle \mathsf{Rule}, \mathcal{T} \rangle} [c\restriction_0][\pi\restriction_n]\phi \text{ and } \sigma \models_{\langle \mathsf{Rule}, \mathcal{T} \rangle} \bigwedge_\rho [apply(\rho, c; \pi)\restriction_{n-1}]\phi.$$

Assume $c \in \mathsf{BasicAction}$ and furthermore assume that $\langle c; \pi, \sigma \rangle \rightarrow_{execute} \langle \pi, \sigma_1 \rangle$ is a transition in $\mathsf{Trans}_{\mathcal{A}}$, i.e., $\kappa(\mathcal{C}_r^{\mathcal{A}}(c\restriction_0, \sigma)) = \{\sigma_1\}$ by Definition 7.14. Let ρ range over $applicable(\mathsf{Rule}, c; \pi)$. Now, observe the following by Definition 7.14:

$$\kappa(\mathcal{C}_r^{\mathcal{A}}(c; \pi\restriction_n, \sigma)) = \kappa(\mathcal{C}_r^{\mathcal{A}}(\pi\restriction_n, \sigma_1)) \cup \bigcup_\rho \kappa(\mathcal{C}_r^{\mathcal{A}}(apply(\rho, c; \pi)\restriction_{n-1}, \sigma)). \quad (7.2)$$

If $c \in \mathsf{AbstractPlan}$ or if a transition of the form $\langle c; \pi, \sigma \rangle \rightarrow_{execute} \langle \pi, \sigma_1 \rangle$ is not derivable, the first term of the right-hand side of (7.2) is empty.

($\Rightarrow$) Assume $\sigma \models_{\mathsf{Rule}} [c; \pi\restriction_n]\phi$, i.e., by Definition 7.16 $\forall \sigma' \in \mathcal{O}_r^{\mathcal{A}}(c; \pi\restriction_n, \sigma) : \sigma' \models_{\mathsf{Rule}} \phi$, i.e., by Definition 7.15:

$$\forall \sigma' \in \kappa(\mathcal{C}_r^{\mathcal{A}}(c; \pi\restriction_n, \sigma)) : \sigma' \models_{\mathsf{Rule}} \phi. \quad (7.3)$$

To prove: (A) $\sigma \models_{\mathsf{Rule}} [c\restriction_0][\pi\restriction_n]\phi$ and (B) $\sigma \models_{\mathsf{Rule}} \bigwedge_\rho [apply(\rho, c; \pi)\restriction_{n-1}]\phi$.
(A) If $c \in \mathsf{AbstractPlan}$ or if a transition of the form $\langle c; \pi, \sigma \rangle \rightarrow_{execute} \langle \pi, \sigma_1 \rangle$ is not derivable, the desired result follows immediately from axiom (PRDL2) or an analogous proposition for non executable basic actions. If $c \in \mathsf{BasicAction}$, we have the following from Definitions 7.16 and 7.15.

$$\begin{array}{rcl} \sigma \models_{\mathsf{Rule}} [c\restriction_0][\pi\restriction_n]\phi & \Leftrightarrow & \forall \sigma' \in \mathcal{O}_r^{\mathcal{A}}(c\restriction_0, \sigma) : \sigma' \models_{\mathsf{Rule}} [\pi\restriction_n]\phi \\ & \Leftrightarrow & \forall \sigma' \in \mathcal{O}_r^{\mathcal{A}}(c\restriction_0, \sigma) : \forall \sigma'' \in \mathcal{O}_r^{\mathcal{A}}(\pi\restriction_n, \sigma') : \sigma'' \models_{\mathsf{Rule}} \phi \\ & \Leftrightarrow & \forall \sigma' \in \kappa(\mathcal{C}_r^{\mathcal{A}}(c\restriction_0, \sigma)) : \forall \sigma'' \in \kappa(\mathcal{C}_r^{\mathcal{A}}(\pi\restriction_n, \sigma')) : \\ & & \sigma'' \models_{\mathsf{Rule}} \phi \\ & \Leftrightarrow & \forall \sigma'' \in \kappa(\mathcal{C}_r^{\mathcal{A}}(\pi\restriction_n, \sigma_1)) : \sigma'' \models_{\mathsf{Rule}} \phi \end{array} \quad (7.4)$$

From (7.2), we have that $\kappa(\mathcal{C}_r^{\mathcal{A}}(\pi\restriction_n, \sigma_1)) \subseteq \kappa(\mathcal{C}_r^{\mathcal{A}}(c; \pi\restriction_n, \sigma))$. From this and assumption (7.3), we can now conclude the desired result (7.4).

(B) Let $c \in (\mathsf{BasicAction} \cup \mathsf{AbstractPlan})$ and let $\rho \in \mathit{applicable}(\mathsf{Rule}, c; \pi)$. Then we want to prove $\sigma \models_{\mathsf{Rule}} [\mathit{apply}(\rho, c; \pi)\restriction_{n-1}]\phi$. From Definitions 7.16 and 7.15, we have the following.

$$\begin{array}{lcl} \sigma \models_{\mathsf{Rule}} [\mathit{apply}(\rho, c; \pi)\restriction_{n-1}]\phi & \Leftrightarrow & \forall \sigma' \in \mathcal{O}_r^{\mathcal{A}}(\mathit{apply}(\rho, c; \pi)\restriction_{n-1}, \sigma) : \sigma' \models_{\mathsf{Rule}} \phi \\ & \Leftrightarrow & \forall \sigma' \in \kappa(\mathcal{C}_r^{\mathcal{A}}(\mathit{apply}(\rho, c; \pi)\restriction_{n-1}, \sigma)) : \\ & & \sigma' \models_{\mathsf{Rule}} \phi \end{array} \tag{7.5}$$

From (7.2), we have that $\kappa(\mathcal{C}_r^{\mathcal{A}}(\mathit{apply}(\rho, c; \pi)\restriction_{n-1}, \sigma)) \subseteq \kappa(\mathcal{C}_r^{\mathcal{A}}(c; \pi\restriction_n, \sigma))$. From this and assumption (7.3), we can now conclude the desired result (7.5).

($\Leftarrow$) Assume $\sigma \models_{\mathsf{Rule}} [c\restriction_0][\pi\restriction_n]\phi$ and $\sigma \models_{\mathsf{Rule}} \bigwedge_\rho [\mathit{apply}(\rho, c; \pi)\restriction_{n-1}]\phi$, i.e., $\forall \sigma' \in \kappa(\mathcal{C}_r^{\mathcal{A}}(\pi\restriction_n, \sigma_1)) : \sigma' \models_{\mathsf{Rule}} \phi$ (7.4) and $\forall \sigma' \in \kappa(\mathcal{C}_r^{\mathcal{A}}(\mathit{apply}(\rho, c; \pi)\restriction_{n-1}, \sigma)) : \sigma' \models_{\mathsf{Rule}} \phi$ (7.5).
To prove: $\sigma \models_{\mathsf{Rule}} [c; \pi\restriction_n]\phi$, i.e., $\forall \sigma' \in \kappa(\mathcal{C}_r^{\mathcal{A}}(c; \pi\restriction_n, \sigma)) : \sigma' \models_{\mathsf{Rule}} \phi$ (7.3). If $c \in \mathsf{AbstractPlan}$ or if a transition of the form $\langle c; \pi, \sigma\rangle \rightarrow_{\mathit{execute}} \langle \pi, \sigma_1\rangle$ is not derivable, we have that $\kappa(\mathcal{C}_r^{\mathcal{A}}(c; \pi\restriction_n, \sigma)) = \bigcup_\rho \kappa(\mathcal{C}_r^{\mathcal{A}}(\mathit{apply}(\rho, c; \pi)\restriction_{n-1}, \sigma))$ (7.2). From this and the assumption, we have the desired result.

If $c \in \mathsf{BasicAction}$ and a transition of the form $\langle c; \pi, \sigma\rangle \rightarrow_{\mathit{execute}} \langle \pi, \sigma_1\rangle$ is derivable, we have (7.2). From this and the assumption, we again have the desired result. □

7.4.2 Completeness

In order to prove completeness of the axiom system, we first prove Proposition 7.1, which says that any formula from $\mathcal{L}_{\mathsf{PRDL}}$ can be rewritten into an equivalent formula where all restriction parameters are 0. This proposition is proven by induction on the size of formulas. The size of a formula is defined by means of the function $\mathit{size} : \mathcal{L}_{\mathsf{PRDL}} \rightarrow \mathbb{N}^3$. This function takes a formula from $\mathcal{L}_{\mathsf{PRDL}}$ and yields a triple $\langle x, y, z\rangle$, where x roughly corresponds to the sum of the restriction parameters occurring in the formula, y roughly corresponds to the sum of the length of plans in the formula and z is the length of the formula. The idea is that the size of a formula is 0 if all restriction parameters are 0. In order to make the induction technically possible, we however also need to incorporate the length of plans and of the formula into the function *size*. This is explained further after the definition of the function.

Definition 7.18 *(size)* Let the following be a lexicographic ordering on tuples $\langle x, y, z\rangle \in \mathbb{N}^3$:

$\langle x_1, y_1, z_1\rangle < \langle x_2, y_2, z_2\rangle$ iff
$x_1 < x_2$ or ($x_1 = x_2$ and $y_1 < y_2$) or ($x_1 = x_2$ and $y_1 = y_2$ and $z_1 < z_2$).

Let max be a function yielding the maximum of two tuples from $\mathbb{N}^3$ and let f and s respectively be functions yielding the first and second element of a tuple. Let l be a function yielding the number of symbols of a syntactic entity and let $l(\epsilon) = 0$. The function $size : \mathcal{L}_{\mathsf{PRDL}} \to \mathbb{N}^3$ is then as defined below.

$$\begin{array}{lcl}
size(p) & = & \langle 0, 0, l(p)\rangle \\
size([\pi\restriction_n]\phi) & = & \begin{cases} \langle n + f(size(\phi)), l(\pi) + s(size(\phi)), l([\pi\restriction_n]\phi)\rangle & \text{if } n > 0 \\ \langle f(size(\phi)), s(size(\phi)), l([\pi\restriction_n]\phi)\rangle & \text{otherwise} \end{cases} \\
size(\neg\phi) & = & \langle f(size(\phi)), s(size(\phi)), l(\neg\phi)\rangle \\
size(\phi \wedge \phi') & = & \langle f(max(size(\phi), size(\phi'))), \\
 & & \quad s(max(size(\phi), size(\phi'))), l(\phi \wedge \phi')\rangle
\end{array}$$

Note that when calculating the plan length of a formula $[\pi\restriction_n]\phi$, i.e., the second element of the tuple $size([\pi\restriction_n]\phi)$, the length of π is added to the length of the plans in ϕ in case $n > 0$. If however $n = 0$ or $n = -1$, the length of π is *not* added to the length of the plans in ϕ and $s(size(\phi))$ is simply returned. This definition of the function $size$ results in the fact that a formula ϕ in which all restriction parameters are 0 (or -1), will satisfy $size(\phi) = \langle 0, 0, l(\phi)\rangle$. Further, this definition gives us that $size([c\restriction_0][\pi\restriction_n]\phi)$ is smaller than $size([c; \pi\restriction_n]\phi)$, which is needed in the proof of Lemma 7.1, which will be used in the proof of Proposition 7.1.

Clause (7.7) of Lemma 7.1 specifies that the right-hand side of axiom (PRDL4) is smaller than the left-hand side. This axiom will usually be used by applying it from left to right to prove a formula such as $[\pi\restriction_n]\phi$. Intuitively, the fact that the formula will get "smaller" as specified through the function $size$, suggests convergence of the deduction process.

Lemma 7.1 Let $\phi \in \mathcal{L}_{\mathsf{PRDL}}$, let $c \in (\mathsf{BasicAction} \cup \mathsf{AbstractPlan})$, let ρ range over $applicable(\mathsf{Rule}, c; \pi)$ and let $n > 0$. The following then holds.

$$size(\phi) < size([\epsilon\restriction_n]\phi) \tag{7.6}$$

$$size([c\restriction_0][\pi\restriction_n]\phi \wedge \bigwedge_{\rho}[apply(\rho, c; \pi)\restriction_{n-1}]\phi) < size([c; \pi\restriction_n]\phi) \tag{7.7}$$

$$size(\phi) < size(\phi \wedge \phi') \tag{7.8}$$

$$size(\phi') < size(\phi \wedge \phi') \tag{7.9}$$

Proof: First, we prove (7.6). From Definition 7.18, we have:

$$size([\epsilon\restriction_n]\phi) = \langle n + f(size(\phi)), s(size(\phi)), l([\epsilon\restriction_n]\phi)\rangle.$$

This is bigger than $size(\phi)$.

Now we prove (7.7). We have the following from Definition 7.18, using that $n > 0$:

$$\begin{array}{ll} size([c;\pi\restriction_n]\phi) & = \langle n + f(size(\phi)), l(c;\pi) + s(size(\phi)), l([c;\pi\restriction_n]\phi)\rangle, \\ size([c\restriction_0][\pi\restriction_n]\phi) & = \langle n + f(size(\phi)), l(\pi) + s(size(\phi)), l([c\restriction_0][\pi\restriction_n]\phi)\rangle, \\ size([apply(\rho,c;\pi)\restriction_{n-1}]\phi) & = \langle (n-1) + f(size(\phi)), l(apply(\rho,c;\pi)) \\ & \quad + s(size(\phi)), l([apply(\rho,c;\pi)\restriction_{n-1}]\phi)\rangle. \end{array}$$

Let $F = [c\restriction_0][\pi\restriction_n]\phi$ and $S = [apply(\rho,c;\pi)\restriction_{n-1}]\phi$. Then, $max(size(F), size(S)) = size(F)$ for any plan revision rule ρ. Thus, $size(F \wedge \bigwedge_\rho S) = \langle n + f(size(\phi)), l(\pi) + s(size(\phi)), l(F \wedge \bigwedge_\rho S)\rangle$, which is smaller than $size([c;\pi\restriction_n]\phi)$, yielding the desired result.

Finally, we prove (7.8) and (7.9). First, we show that $size(\phi) < size(\phi \wedge \phi')$, which we will refer to by R. We thus have to show:

$$\begin{array}{l} \langle f(size(\phi)), s(size(\phi)), l(\phi)\rangle < \\ \qquad \langle f(max(size(\phi), size(\phi'))), s(max(size(\phi), size(\phi'))), l(\phi \wedge \phi')\rangle. \end{array}$$

If $f(size(\phi)) < f(max(size(\phi), size(\phi')))$, we have R. If $f(size(\phi)) = f(max(size(\phi), size(\phi')))$ and $s(size(\phi)) < s(max(size(\phi), size(\phi')))$, we again have R. If $s(size(\phi)) = s(max(size(\phi), size(\phi')))$, we also have R, because $l(\phi) < l(\phi \wedge \phi')$. Covering all cases, this yields the desired result. The same line of reasoning can be applied to show $size(\phi') < size(\phi \wedge \phi')$. □

Now we can formulate and prove the following proposition.

Proposition 7.1 Any formula $\phi \in \mathcal{L}_{\mathsf{PRDL}}$ can be rewritten into an equivalent formula ϕ_{PDL} where all restriction parameters are 0, i.e.:

$$\forall \phi \in \mathcal{L}_{\mathsf{PRDL}} : \exists \phi_{\mathsf{PDL}} \in \mathcal{L}_{\mathsf{PRDL}} : size(\phi_{\mathsf{PDL}}) = \langle 0, 0, l(\phi_{\mathsf{PDL}})\rangle \text{ and } \vdash_{\mathsf{Rule}} \phi \leftrightarrow \phi_{\mathsf{PDL}}.$$

Proof: The fact that a formula ϕ has the property that it can be rewritten as specified in the proposition, will be denoted by $\mathsf{PDL}(\phi)$ for reasons that will become clear in the sequel. The proof is by induction on $size(\phi)$.

- $\phi \equiv p$
  $size(p) = \langle 0, 0, l(p)\rangle$ and let $p_{\mathsf{PDL}} = p$, then $\mathsf{PDL}(p)$.
- $\phi \equiv [\pi\restriction_n]\phi'$
  If $n = -1$, we have that $[\pi\restriction_n]\phi'$ is equivalent with $\top$ (PRDL1). As $\mathsf{PDL}(\top)$, we also have $\mathsf{PDL}([\pi\restriction_n]\phi')$ in this case.

  Let $n = 0$. We then have that $size([\pi\restriction_n]\phi') = \langle f(size(\phi')), s(size(\phi')), l([\pi\restriction_n]\phi')\rangle$ is greater than $size(\phi') = \langle f(size(\phi')), s(size(\phi')), l(\phi')\rangle$. By induction, we then have $\mathsf{PDL}(\phi')$, i.e., ϕ' can be rewritten into an equivalent formula ϕ'_{PDL}, such that

$size(\phi'_{\mathsf{PDL}}) = \langle 0, 0, l(\phi'_{\mathsf{PDL}})\rangle$. As $size([\pi\restriction_n]\phi'_{\mathsf{PDL}}) = \langle 0, 0, l([\pi\restriction_n]\phi'_{\mathsf{PDL}})\rangle$, we have $\mathsf{PDL}([\pi\restriction_n]\phi'_{\mathsf{PDL}})$ and therefore $\mathsf{PDL}([\pi\restriction_n]\phi')$.

Let $n > 0$. Let $\pi = \epsilon$. By Lemma 7.1, we have $size(\phi') < size([\epsilon\restriction_n]\phi')$. Therefore, by induction, $\mathsf{PDL}(\phi')$. As $[\epsilon\restriction_n]\phi'$ is equivalent with ϕ' by axiom (PRDL3), we also have $\mathsf{PDL}([\epsilon\restriction_n]\phi')$. Now let $\pi = c;\pi'$ and let $L = [c;\pi'\restriction_n]\phi'$ and $R = [c\restriction_0][\pi'\restriction_n]\phi' \wedge \bigwedge_\rho [apply(\rho, c;\pi')\restriction_{n-1}]\phi'$. By Lemma 7.1, we have that $size(R) < size(L)$. Therefore, by induction, we have $\mathsf{PDL}(R)$. As R and L are equivalent by axiom (PRDL4), we also have $\mathsf{PDL}(L)$, yielding the desired result.

- $\phi \equiv \neg\phi'$
  We have that $size(\neg\phi') = \langle f(size(\phi')), s(size(\phi')), l(\neg\phi')\rangle$, which is greater than $size(\phi')$. By induction, we thus have $\mathsf{PDL}(\phi')$ and $size(\phi'_{\mathsf{PDL}}) = \langle 0, 0, l(\phi'_{\mathsf{PDL}})\rangle$. Then, $size(\neg\phi'_{\mathsf{PDL}}) = \langle 0, 0, l(\neg\phi'_{\mathsf{PDL}})\rangle$ and thus $\mathsf{PDL}(\neg\phi'_{\mathsf{PDL}})$ and therefore $\mathsf{PDL}(\neg\phi')$.

- $\phi \equiv \phi' \wedge \phi''$
  By Lemma 7.1, we have $size(\phi') < size(\phi' \wedge \phi'')$ and $size(\phi'') < size(\phi' \wedge \phi'')$. Therefore, by induction, $\mathsf{PDL}(\phi')$ and $\mathsf{PDL}(\phi'')$ and therefore $size(\phi'_{\mathsf{PDL}}) = \langle 0, 0, l(\phi'_{\mathsf{PDL}})\rangle$ and $size(\phi''_{\mathsf{PDL}}) = \langle 0, 0, l(\phi''_{\mathsf{PDL}})\rangle$. Then, $size(\phi'_{\mathsf{PDL}} \wedge \phi''_{\mathsf{PDL}}) = \langle 0, 0, l(\phi'_{\mathsf{PDL}} \wedge \phi''_{\mathsf{PDL}})\rangle$ and therefore $size((\phi' \wedge \phi'')_{\mathsf{PDL}}) = \langle 0, 0, l((\phi' \wedge \phi'')_{\mathsf{PDL}})\rangle$ and we can conclude $\mathsf{PDL}((\phi' \wedge \phi'')_{\mathsf{PDL}})$ and thus $\mathsf{PDL}(\phi' \wedge \phi'')$.

□

Although structural induction is not possible for plans in general, it *is* possible if we only consider action execution, i.e., if the restriction parameter is 0. This is specified in the following proposition, from which we can conclude that a formula ϕ with $size(\phi) = \langle 0, 0, l(\phi)\rangle$ satisfies all standard PDL properties.

Proposition 7.2 *(sequential composition)* Let $\mathsf{Rule} \subseteq \mathcal{R}$ be a finite set of plan revision rules. The following is then derivable in the axiom system $\mathsf{AS}_{\mathsf{Rule}}$.

$$\vdash_{\mathsf{Rule}} [\pi_1;\pi_2\restriction_0]\phi \leftrightarrow [\pi_1\restriction_0][\pi_2\restriction_0]\phi$$

Proof: If $\pi_1 = \epsilon$, we have $[\pi_2\restriction_0] \leftrightarrow [\epsilon\restriction_0][\pi_2\restriction_0]\phi$ by axiom (PRDL3). Otherwise, let $c_i \in (\mathsf{BasicAction} \cup \mathsf{AbstractPlan})$ for $i \geq 1$, let $\pi_1 = c_1;\ldots;c_n$, with $n \geq 1$. Through repeated application of axiom (PRDL4), first from left to right and then from right to left (also using axiom (PRDL1) to eliminate the rule application

part of the axiom), we derive the desired result.[9]

$$\begin{array}{lcl}
[\pi_1;\pi_2\restriction_0]\phi & \leftrightarrow & [c_1;\ldots;c_n;\pi_2\restriction_0]\phi \\
& \leftrightarrow & [c_1\restriction_0][c_2;\ldots;c_n;\pi_2\restriction_0]\phi \\
& \leftrightarrow & \ldots \\
& \leftrightarrow & [c_1\restriction_0][c_2\restriction_0]\ldots[c_n\restriction_0][\pi_2\restriction_0]\phi \\
& \leftrightarrow & [c_1;c_2\restriction_0][c_3\restriction_0]\ldots[c_n\restriction_0][\pi_2\restriction_0]\phi \\
& \leftrightarrow & \ldots \\
& \leftrightarrow & [c_1;\ldots;c_n\restriction_0][\pi_2\restriction_0]\phi \\
& \leftrightarrow & [\pi_1\restriction_0][\pi_2\restriction_0]\phi
\end{array}$$

□

Theorem 7.2 *(completeness)* Let $\phi \in \mathcal{L}_{\mathsf{PRDL}}$ and let $\mathsf{Rule} \subseteq \mathcal{R}$ be a finite set of plan revision rules. Then the axiom system $\mathsf{AS}_{\mathsf{Rule}}$ is complete, i.e.:

$$\models_{\mathsf{Rule}} \phi \;\Rightarrow \vdash_{\mathsf{Rule}} \phi.$$

Proof: Let $\phi \in \mathcal{L}_{\mathsf{PRDL}}$. By Proposition 7.1 we have that a formula ϕ_{PDL} exists such that $\vdash_{\mathsf{Rule}} \phi \leftrightarrow \phi_{\mathsf{PDL}}$ and $size(\phi_{\mathsf{PDL}}) = \langle 0, 0, l(\phi_{\mathsf{PDL}})\rangle$ and therefore by soundness of $\mathsf{AS}_{\mathsf{Rule}}$ also $\models_{\mathsf{Rule}} \phi \leftrightarrow \phi_{\mathsf{PDL}}$. Let ϕ_{PDL} be a formula with these properties.

$$\begin{array}{lcll}
\models_{\mathsf{Rule}} \phi & \Leftrightarrow & \models_{\mathsf{Rule}} \phi_{\mathsf{PDL}} & (\models_{\mathsf{Rule}} \phi \leftrightarrow \phi_{\mathsf{PDL}}) \\
& \Rightarrow & \vdash_{\mathsf{Rule}} \phi_{\mathsf{PDL}} & (\text{completeness of } \mathsf{PDL}) \\
& \Leftrightarrow & \vdash_{\mathsf{Rule}} \phi & (\vdash_{\mathsf{Rule}} \phi \leftrightarrow \phi_{\mathsf{PDL}})
\end{array}$$

The second step in this proof needs some justification. The general idea is that all PDL axioms and rules are applicable to a formula ϕ_{PDL} and moreover, these axioms and rules are contained in our axiom system $\mathsf{AS}_{\mathsf{Rule}}$. As PDL is complete, we have $\models_{\mathsf{Rule}} \phi_{\mathsf{PDL}} \Rightarrow \vdash_{\mathsf{Rule}} \phi_{\mathsf{PDL}}$. There are however some subtleties to be considered, as our action language is not exactly the same as the action language of PDL, nor is it a subset (at first sight).

The action language of PDL is built using basic actions, sequential composition, test, non-deterministic choice and iteration. The action language of PRDL is built using basic actions, abstract plans, empty plans and sequential composition. If we for the moment disregard abstract plans and empty plans, the language PRDL is a subset of the language PDL. If we take the subset of PDL axioms and rules dealing with formulas in this subset, this axiom system should be complete with respect to these formulas.

The action language of full PRDL however also contains abstract plans and empty plans. The question is, how these should be axiomatized such that we

[9] We use the notation $\phi_1 \leftrightarrow \phi_2 \leftrightarrow \phi_3 \leftrightarrow \ldots$, which should be read as a shorthand for $\phi_1 \leftrightarrow \phi_2$ and $\phi_2 \leftrightarrow \phi_3$ and ... This notation will also be used in the sequel.

obtain a complete axiomatization. In order to answer this question, we make the following observation. In a formula ϕ_{PDL}, abstract and empty plans can only occur with a 0 restriction parameter by definition. Further, the semantics of a formula $[p\restriction_0]\phi_{\mathsf{PDL}}$ where p is an abstract plan, is similar to the semantics of the `fail` statement of (an extended version of) PDL. The set of states resulting from "execution" of both statements is empty.[10] The semantics of a formula $[\epsilon\restriction_0]\phi_{\mathsf{PDL}}$ is similar to the semantics of the `skip` statement of PDL. The set of states resulting from the execution of both statements in a state σ is $\{\sigma\}$,[11] i.e., the semantics is the identity relation. The action language of PRDL can thus be considered to be a subset of the action language of PDL, where $p\restriction_0$ and $\epsilon\restriction_0$ correspond respectively to `fail` and `skip`.

Now, `fail` and `skip` are not axiomatized in the basic axiom system of PDL. These statements are however defined as $\mathbf{0}?$ and $\mathbf{1}?$ respectively and the test statement *is* axiomatized: $[\psi?]\phi \leftrightarrow (\psi \rightarrow \phi)$. We now fill in $\mathbf{0}$ and $\mathbf{1}$ for ψ in this axiom, which gives us the following.

$$\begin{array}{lcll} [\mathbf{0}?]\phi \leftrightarrow (\mathbf{0} \rightarrow \phi) & \Leftrightarrow & [\mathbf{0}?]\phi & \Leftrightarrow \quad [\texttt{fail}]\phi \\ [\mathbf{1}?]\phi \leftrightarrow (\mathbf{1} \rightarrow \phi) & \Leftrightarrow & [\mathbf{1}?]\phi \leftrightarrow \phi & \Leftrightarrow \quad [\texttt{skip}]\phi \leftrightarrow \phi \end{array}$$

The statements `fail` and `skip` are thus implicitly axiomatized through the axiomatization of the test. For our axiom system to be complete for formulas ϕ_{PDL}, it should thus contain the PDL axioms and rules that are applicable to these formulas, that is, the axiom for sequential composition, the axioms for `fail` and `skip` as stated above, the axiom for distribution of box over implication and the rules (MP) and (GEN). The latter three are explicitly contained in $\mathsf{AS}_{\mathsf{Rule}}$. The axiom for sequential composition is derivable in the system $\mathsf{AS}_{\mathsf{Rule}}$ for formulas ϕ_{PDL}, by Proposition 7.2. Axiom (PRDL2) for $p\restriction_0$ corresponds with the axiom for `fail`. The axiom for $\epsilon\restriction_0$, corresponding with the axiom for `skip`, is an instantiation of axiom (PRDL3). Axiom (PRDL3), i.e., the more general version of $[\epsilon\restriction_0]\phi \leftrightarrow \phi$, is needed in the proof of Proposition 7.1, which is used elsewhere in this completeness proof. □

We conclude with a remark with respect to axiom (PRDL3). In the proof above, we explained that the semantics of $\epsilon\restriction_0$ and `skip` are equivalent. As it turns out (see Proposition 7.3), $[\epsilon\restriction_0]\phi$ is equivalent with $[\epsilon\restriction_n]\phi$, as can be proven from axiom (PRDL3), which is thus also equivalent with `skip`.

Proposition 7.3 *(empty plan)* Let $\mathsf{Rule} \subseteq \mathcal{R}$ be a finite set of plan revision rules. The following is then derivable in the axiom system $\mathsf{AS}_{\mathsf{Rule}}$.

$$\vdash_{\mathsf{Rule}} [\epsilon\restriction_0]\phi \leftrightarrow [\epsilon\restriction_n]\phi \text{ with } 0 \leq n$$

[10] An abstract plan p cannot be executed directly, it can only be transformed using plan revision rules. The restriction parameter is however 0, so no plan revision rules may be applied and the set $\mathcal{O}_r^{\mathcal{A}}([p\restriction_0]\phi)(\sigma) = \emptyset$ for all $\mathcal{A}$ and σ.

[11] $\mathcal{C}_r^{\mathcal{A}}([\epsilon\restriction_0]\phi_{\mathsf{PDL}})(\sigma) = \{\sigma\} = \kappa(\mathcal{C}_r^{\mathcal{A}}([\epsilon\restriction_0]\phi_{\mathsf{PDL}})(\sigma)) = \mathcal{O}_r^{\mathcal{A}}([\epsilon\restriction_0]\phi_{\mathsf{PDL}})(\sigma)$

Proof:

1.	$[\epsilon\restriction_n][\epsilon\restriction_0]\phi \leftrightarrow [\epsilon\restriction_0]\phi$	(PRDL3)
2.	$[\epsilon\restriction_0]\phi \leftrightarrow \phi$	(PRDL3)
3.	$[\epsilon\restriction_n][\epsilon\restriction_0]\phi \leftrightarrow [\epsilon\restriction_n]\phi$	2, (GEN), (PDL)
4.	$[\epsilon\restriction_0]\phi \leftrightarrow [\epsilon\restriction_n]\phi$	1, 3, (PL)

□

7.5 Proving Properties of Non-Restricted Plans

In Sections 7.3 and 7.4 we have presented a logic for restricted plans with sound and complete axiomatization. This means that it should be possible to construct a proof for, e.g., a formula $[a; b\restriction_3]\phi$ if and only if it is true for a given agent. This might be considered an interesting result, but our ultimate aim is to prove properties of non-restricted 3APL plans.

The semantics of restricted plans is closely related to the semantics of non-restricted plans. Using this relation, we will show how the proof system for restricted plans can be extended to a proof system for non-restricted plans. Then we will discuss the usability of this system, using examples.

7.5.1 From Restricted to Non-Restricted Plans

We first add the following clause to the language $\mathcal{L}_{\mathsf{PRDL}}$ (Definition 7.13),[12] yielding a language that we will call $\mathcal{L}_{\mathsf{PRDL}^+}$: if $\phi \in \mathcal{L}_{\mathsf{PRDL}^+}$ and $\pi \in \mathsf{Plan}$, then $[\pi]\phi \in \mathcal{L}_{\mathsf{PRDL}^+}$. By means of this construct, we can thus specify properties of non-restricted plans. We define the semantics of this construct in terms of the operational semantics of non-restricted plans as follows.

Definition 7.19 *(semantics of* PRDL^+*)* Let $\mathcal{A}$ be a 3APL agent (Definition 7.4). The semantics of formulas not of the form $[\pi]\phi$ with $\phi \in \mathcal{L}_{\mathsf{PRDL}^+}$ is as in Definition 7.16. The semantics of formulas of the form $[\pi]\phi$ is as defined below.

$$\sigma \models_{\mathcal{A}} [\pi]\phi \Leftrightarrow \forall \sigma' \in \mathcal{O}^{\mathcal{A}}(\pi)(\sigma) : \sigma' \models_{\mathcal{A}} \phi$$

This definition thus takes the operational semantics of non-restricted plans to define the semantics of constructs of the form $[\pi]\phi$. In the following proposition, we relate the operational semantics of plans and the operational semantics of restricted plans.

Proposition 7.4

$$\bigcup_{n \in \mathbb{N}} \mathcal{O}_r(\pi\restriction_n)(\sigma) = \mathcal{O}(\pi)(\sigma)$$

[12] Replacing each occurrence of $\mathcal{L}_{\mathsf{PRDL}}$ in this definition by $\mathcal{L}_{\mathsf{PRDL}^+}$.

Proof: Immediate from Definitions 7.15, 7.14, 7.11 and 7.10. □

From this proposition, we have the following corollary, which shows how the construct $[\pi\restriction_n]\phi$ is related to the construct $[\pi]\phi$.

Corollary 7.1

$$\begin{array}{rcl} \forall n \in \mathbb{N} : \sigma \models_{\mathcal{A}} [\pi\restriction_n]\phi & \Leftrightarrow & \forall \sigma' \in \mathcal{O}^{\mathcal{A}}(\pi)(\sigma) : \sigma' \models_{\mathcal{A}} \phi \\ & \Leftrightarrow & \sigma \models_{\mathcal{A}} [\pi]\phi \end{array}$$

Proof: Immediate from Proposition 7.4, Definition 7.16 and Definition 7.19. □

From this corollary, we can conclude that we can prove a property of the form $[\pi]\phi$ by proving $\forall n \in \mathbb{N} : \vdash_{\mathsf{Rule}} [\pi\restriction_n]\phi$, using the system for restricted plans. This idea can be captured in a proof rule as follows.

Definition 7.20 *(proof rule for non-restricted plans)*

$$\frac{[\pi\restriction_n]\phi,\ n \in \mathbb{N}}{[\pi]\phi}$$

This rule should be read as having an infinite number of premises, i.e., $[\pi\restriction_0]\phi$, $[\pi\restriction_1]\phi$, $[\pi\restriction_2]\phi$, ... (see also [Harel et al., 2000]). Deriving a formula $[\pi]\phi$ using this infinitary rule thus requires infinitely many premises to have been previously derived.

The rule is sound by corollary 7.1. The system $\mathsf{AS}_{\mathsf{Rule}}$ for restricted plans (Definition 7.17) taken together with the rule above, is a complete axiom system for PRDL^+: if $[\pi]\phi$ is true then each of the premises of the rule is true (corollary 7.1) and each of these premises can be proven by completeness of $\mathsf{AS}_{\mathsf{Rule}}$. The notion of a proof in this case is however non-standard, as a proof can be infinite. This completeness result is therefore theoretical, and putting the system to use in this way is obviously problematic.

One way to try to deal with this problem is the following. The idea is that properties of the form $\forall n \in \mathbb{N} : \vdash_{\mathsf{Rule}} [\pi\restriction_n]\phi$ can be proven by *induction* on n, rather than proving $[\pi\restriction_n]\phi$ for each n. If we can prove $[\pi\restriction_0]\phi$ and $\forall n \in \mathbb{N} : ([\pi\restriction_n]\phi \vdash_{\mathsf{Rule}} [\pi\restriction_{n+1}]\phi)$, we can conclude the desired property. In the next section we will illustrate how this could be done, using examples. The examples however show that it is not obvious that this kind of induction can be applied in all cases.

7.5.2 Examples

Example 7.2 Let $\mathcal{A}$ be an agent with one plan revision rule, i.e., $\mathsf{Rule} = \{a; b \rightsquigarrow c\}$ and let $\mathcal{T}$ be such that $[a\restriction_0]\phi$, $[b\restriction_0]\phi$ and $[c\restriction_0]\phi$. We now want to prove that $\forall n : [a; b\restriction_n]\phi$. We have $[a; b\restriction_0]\phi$ by using that this is equivalent to

$[a\restriction_0][b\restriction_0]\phi$ by Proposition 7.2. The latter formula can be derived by applying (GEN) to $[b\restriction_0]\phi$. We prove $\forall n \in \mathbb{N} : ([a;b\restriction_n]\phi \vdash_{\mathsf{Rule}} [a;b\restriction_{n+1}]\phi)$ by taking an arbitrary n and proving that $[a;b\restriction_n]\phi \vdash_{\mathsf{Rule}} [a;b\restriction_{n+1}]\phi$. Using (PRDL4) and (PRDL3), we have the following equivalences.

$$\begin{array}{lcll} [a;b\restriction_n]\phi & \leftrightarrow & [a\restriction_0][b\restriction_n]\phi & \wedge \quad [c\restriction_{n-1}]\phi \\ & \leftrightarrow & [a\restriction_0][b\restriction_0][\epsilon\restriction_n]\phi & \wedge \quad [c\restriction_0][\epsilon\restriction_{n-1}]\phi \\ & \leftrightarrow & [a\restriction_0][b\restriction_0]\phi & \wedge \quad [c\restriction_0]\phi \end{array}$$

Similarly, we have the following equivalences for $[a;b\restriction_{n+1}]\phi$, yielding the desired result.

$$\begin{array}{lcll} [a;b\restriction_{n+1}]\phi & \leftrightarrow & [a\restriction_0][b\restriction_{n+1}]\phi & \wedge \quad [c\restriction_n]\phi \\ & \leftrightarrow & [a\restriction_0][b\restriction_0][\epsilon\restriction_{n+1}]\phi & \wedge \quad [c\restriction_0][\epsilon\restriction_n]\phi \\ & \leftrightarrow & [a\restriction_0][b\restriction_0]\phi & \wedge \quad [c\restriction_0]\phi \end{array}$$

△

Example 7.3 We will prove a property of a very simple 3APL agent using axiom (PRDL4) and induction on the number of plan revision rule applications. Our agent has one plan revision rule: $\mathsf{Rule} = \{a \rightsquigarrow a;a\}$. Furthermore, assume that $\mathcal{T}$ is defined such that $[a\restriction_0]\phi$. We want to prove the following: $\forall n \in \mathbb{N} : [a\restriction_n]\phi$. In order to prove the desired result by induction on the number of plan revision rule applications, we thus have to prove $[a\restriction_0]\phi$ and $\forall n \in \mathbb{N} : [a\restriction_n]\phi \vdash_{\mathsf{Rule}} [a\restriction_{n+1}]\phi$. $[a\restriction_0]\phi$ was given. Let a^i denote a sequence of a's of length i, with $a^0 = \epsilon$. The premise of the second conjunct can be rewritten using axiom (PRDL4) as follows.

$$\begin{array}{lcl} [a\restriction_n]\phi & \leftrightarrow & [a\restriction_0]\phi \wedge [(a;a)\restriction_{n-1}]\phi \\ & \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_{n-1}]\phi \wedge [(a;a;a)\restriction_{n-2}]\phi \\ & \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_{n-1}]\phi \wedge [a\restriction_0][(a;a)\restriction_{n-2}]\phi \wedge [(a;a;a;a)\restriction_{n-3}]\phi \\ & \vdots & \\ & \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_{n-1}]\phi \wedge \ldots \wedge [a\restriction_0][(a^n)\restriction_0]\phi \wedge [(a;(a^n))\restriction_0]\phi \end{array}$$

So, in order to prove $[a\restriction_{n+1}]\phi$, we may assume - among other things - $[a\restriction_n]\phi$, $[(a;a)\restriction_{n-1}]\phi$, $[(a;a;a)\restriction_{n-2}]\phi$, …, $[(a;(a^n))\restriction_0]\phi$ (last conjunct of each line). Equivalently, we may thus assume the following.[13]

$$\bigwedge_i [(a;(a^i))\restriction_{n-i}]\phi \qquad \text{for } 0 \leq i \leq n \tag{7.10}$$

[13]Note that $[a\restriction_0][(a^0)\restriction_n]\phi \leftrightarrow [a\restriction_0][\epsilon\restriction_n]\phi$ and $[a\restriction_0][\epsilon\restriction_n]\phi \leftrightarrow [a\restriction_0]\phi$, using axiom (PRDL3).

The consequent, i.e., $[a\restriction_{n+1}]\phi$, can be rewritten using axiom (PRDL4) as below.

$$\begin{array}{rcl}
[a\restriction_{n+1}]\phi & \leftrightarrow & [a\restriction_0]\phi \wedge [(a;a)\restriction_n]\phi \\
& \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_n]\phi \wedge [(a;a;a)\restriction_{n-1}]\phi \\
& \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_n]\phi \wedge [a\restriction_0][(a;a)\restriction_{n-1}]\phi \wedge [(a;a;a;a)\restriction_{n-2}]\phi \\
& \vdots & \\
& \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_n]\phi \wedge \ldots \wedge [a\restriction_0][(a;(a^n))\restriction_0]\phi \wedge [(a;a;(a^n))\restriction_0]\phi
\end{array} \tag{7.11}$$

As $[a\restriction_{n+1}]\phi$ is equivalent to all of the lines on the righthandside of (7.11), we may prove any of these lines, in order to prove the desired result. As it turns out, it is easiest to prove the last line. The reason is that in this case, the last conjunct has a restriction parameter of 0. We can thus use Proposition 7.2 for sequential composition to prove this conjunct as follows.

1.	$[a\restriction_0]\phi$	assumption
2.	$[(a;a;(a^{n-1}))\restriction_0][a\restriction_0]\phi$	1, (GEN)
3.	$[(a;a;(a^{n-1});a)\restriction_0]\phi$	2, Proposition 7.2
4.	$[(a;a;(a^n))\restriction_0]\phi$	3, definition of a^i

Proving the other part of the last line of (7.11), i.e., $\bigwedge_i [a\restriction_0][(a;(a^i))\restriction_{n-i}]\phi$ for $0 \leq i \leq n$, can be done by applying (GEN) to each of the conjuncts of 7.10, yielding the desired result. △

The important thing to note about this example is that rewriting of formulas like $[a\restriction_n]\phi$ using (PRDL4), terminates. This is because the number of rewrite steps is restricted by n. If we would not have this restriction parameter, we might have the following variant of (PRDL4):

$$[c;\pi]\phi \leftrightarrow [c\restriction_0][\pi]\phi \wedge \bigwedge_\rho [apply(\rho, c;\pi)]\phi.^{14}$$

An attempt to proving $[a]\phi$ for an agent with the plan revision rule of Example 7.3 and this "axiom", would however result in infinite regression:

$$\begin{array}{rcl}
[a]\phi & \leftrightarrow & [a\restriction_0]\phi \wedge [a;a]\phi \\
& \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a]\phi \wedge [a;a;a]\phi \\
& \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a]\phi \wedge [a\restriction_0][a;a]\phi \wedge [a;a;a;a]\phi \\
& \vdots &
\end{array}$$

In the example above, we have proven the desired result in our axiom system, using the key axiom (PRDL4). Another way to look at an agent with only the plan revision rule $a \rightsquigarrow a;a$, is by considering the language of plans that is "generated" by this rule. By doing this, a much simpler proof can be obtained.

[14] We use the 0-restriction parameter here to distinguish between rule application and action execution, i.e., $[c;\pi]\phi$ is true, if and only if $[\pi]\phi$ is true after the execution of c and ϕ is true after the plans resulting from the application of the plan revision rules of the agent.

Example 7.4 We take again the agent of Example 7.3, i.e., an agent with one plan revision rule $a \rightsquigarrow a; a$, and with $[a\restriction_0]\phi$. We want to prove again $\forall n \in \mathbb{N} : [a\restriction_n]\phi$. Taking into account the plan revision rule that is given and the initial plan a, one can conclude that the action sequences that can be executed by this agent, are sequences of a of an arbitrary length. Given this, one could instead prove $\forall n \in \mathbb{N}^+ : [a^n\restriction_0]\phi$, where $\mathbb{N}^+$ is the set of positive natural numbers.[15] We prove this by taking an arbitrary n and proving $[a^n\restriction_0]\phi$ for this n.

1.	$[a\restriction_0]\phi$	assumption
2.	$[a\restriction_0][a\restriction_0]\phi$	1, GEN
3.	$[a; a\restriction_0]\phi$	2, Proposition 7.2
	$\vdots$	
	$[a^n\restriction_0]\phi$	

△

Obviously, this proof is much shorter than the proof of Example 7.3. It is however obtained through meta-reasoning about the plan revision rules of the agent. In the desired result $\forall n \in \mathbb{N}^+ : [a^n\restriction_0]\phi$, the restriction parameter is 0. The application of plan revision rules has thus in effect been eliminated from the expression in the object language.

Meta-reasoning could be done in this simple case: the plan revision rule actually generates the language of plans that can be represented by the simple regular expression a^*. Plan revision rules in general however do not only generate languages that can be represented by regular expressions. In particular, rules of the form $p \rightsquigarrow \pi$, where p is an abstract plan, can be compared with parameterless recursive procedures (see also Section 7.6), which can in turn be linked to context-free programs [Harel et al., 2000, Chapter 9]. Furthermore, plan revision rules can have the form $\pi_h \rightsquigarrow \pi_b$, where the head is an arbitrary plan. It is thus not obvious that a meta-argument about the plans generated by the agent can be constructed in the general case. Investigations along these lines are however not within the scope of this chapter and remain for future research.

In the next example, we will use Proposition 7.5 below, in the proof of which we use the following lemma.

Lemma 7.2 Let $\mathsf{Rule} \subseteq \mathcal{R}$ be a finite set of plan revision rules. The following is then derivable in the axiom system $\mathsf{AS}_{\mathsf{Rule}}$.

$$\vdash_{\mathsf{Rule}} [\pi\restriction_n]\phi \rightarrow [\pi\restriction_0]\phi$$

[15] The result $\forall n \in \mathbb{N} : [a\restriction_n]\phi$ that we want to prove specifies that always at least one action a is executed: if $n = 0$, the required result is $[a\restriction_0]\phi$, which specifies the execution of a. The result does not require proving $[\epsilon\restriction_n]\phi$, which would be provable if we would assume ϕ to be valid.

Proof: Let $c_i \in (\mathsf{BasicAction} \cup \mathsf{AbstractPlan})$ for $i \geq 1$ and let $\pi = c_1; \ldots; c_m$, with $m \geq 1$. Through repeated application of axiom (PRDL4), from left to right, then using (PRDL3) to get rid of $[\epsilon\restriction_n]$ and then using Proposition 7.2 for sequential composition with a 0 restriction parameter, we derive the desired result.

$$\begin{array}{lcl}
[\pi\restriction_n]\phi & \leftrightarrow & [c_1; \ldots; c_m\restriction_n]\phi \\
& \rightarrow & [c_1\restriction_0][c_2; \ldots; c_m\restriction_n]\phi \\
& \rightarrow & \ldots \\
& \rightarrow & [c_1\restriction_0][c_2\restriction_0]\ldots[c_m\restriction_0][\epsilon\restriction_n]\phi \\
& \rightarrow & [c_1\restriction_0][c_2\restriction_0]\ldots[c_m\restriction_0]\phi \\
& \rightarrow & [c_1; c_2\restriction_0][c_3\restriction_0]\ldots[c_m\restriction_0]\phi \\
& \rightarrow & \ldots \\
& \rightarrow & [c_1; \ldots; c_m\restriction_0]\phi \\
& \rightarrow & [\pi\restriction_0]\phi
\end{array}$$

□

In the following proposition, we will use some notation that we will first explain. The notation $(\mathsf{PRDL4})_i([\pi\restriction_n]\phi)$, with $0 \leq i \leq n$, denotes the formula that results from rewriting $[\pi\restriction_n]\phi$ using (PRDL4) from left to right, such that all restriction parameters are either 0 or i. Formulas of the form $[\epsilon\restriction_m]\phi$ are replaced by ϕ, using axiom (PRDL3). In this process, (PRDL4) may only be applied to a formula $[\pi\restriction_m]\phi$ if $m > i$.

Take, e.g., the agent of Example 7.3 with $a \rightsquigarrow a; a$ as the only plan revision rule. The formula $(\mathsf{PRDL4})_3([a\restriction_5]\phi)$ then for example denotes the formula $[a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_0]\phi \wedge [a\restriction_0][a; a\restriction_3]\phi \wedge [a; a; a\restriction_3]\phi$, which can be obtained by rewriting the formula $[a\restriction_5]\phi$ as below.

$$\begin{array}{lcl}
[a\restriction_5]\phi & \leftrightarrow & [a\restriction_0][\epsilon\restriction_5]\phi \wedge [a; a\restriction_4]\phi \\
& \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_4]\phi \wedge [a; a; a\restriction_3]\phi \\
& \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_0][\epsilon\restriction_4]\phi \wedge [a\restriction_0][a; a\restriction_3]\phi \wedge [a; a; a\restriction_3]\phi \\
& \leftrightarrow & [a\restriction_0]\phi \wedge [a\restriction_0][a\restriction_0]\phi \wedge [a\restriction_0][a; a\restriction_3]\phi \wedge [a; a; a\restriction_3]\phi
\end{array}$$

The idea is thus, that formulas of the form $[\pi\restriction_m]\phi$ are rewritten until formulas are obtained with i as the restriction parameter. A formula $[\pi\restriction_i]\phi$ may not be rewritten.

Any formula $[\pi\restriction_n]\phi$ can be rewritten into a formula $(\mathsf{PRDL4})_i([\pi\restriction_n]\phi)$ with $0 \leq i \leq n$. An application of (PRDL4) to a formula $[\pi\restriction_m]\phi$ yields two conjuncts (the second of which is again a conjunction). The first conjunct is smaller in plan size than $[\pi\restriction_m]\phi$.[16] Each conjunct of the second conjunct is smaller than $[\pi\restriction_m]\phi$ with respect to the restriction parameter. With each rewrite step, we thus have a decrease either in plan size or in size of the restriction parameter of each resulting conjunct. This can thus continue for each conjunct until either

[16]The second element of $size(F)$, where F denotes the first conjunct, is smaller than the second element of $size([\pi\restriction_m]\phi$.

the plan size (minus the plan size of ϕ) is 0 or the non-zero restriction parameters are equal to i.

Another notation that we will use is $to0(\phi)$, denoting the formula that results from replacing all restriction parameters in ϕ by 0.

Proposition 7.5 *(restriction parameter)* Let $\mathsf{Rule} \subseteq \mathcal{R}$ be a finite set of plan revision rules. The following is then derivable in the axiom system $\mathsf{AS}_{\mathsf{Rule}}$.

$$\vdash_{\mathsf{Rule}} [\pi\restriction_n]\phi \rightarrow [\pi\restriction_i]\phi \text{ with } -1 \leq i \leq n$$

Proof: If $i = -1$, the desired result follows immediately by axiom (PRDL1). We will now prove the result for $i \geq 0$.

1.	$[\pi\restriction_n]\phi \leftrightarrow (\mathsf{PRDL4})_{n-i}([\pi\restriction_n]\phi)$	(PRDL4)
2.	$[\pi\restriction_i]\phi \leftrightarrow (\mathsf{PRDL4})_0([\pi\restriction_i]\phi)$	(PRDL4)
3.	$(\mathsf{PRDL4})_{n-i}([\pi\restriction_n]\phi) \rightarrow to0((\mathsf{PRDL4})_{n-i}([\pi\restriction_n]\phi))$	Lemma 7.2
4.	$to0((\mathsf{PRDL4})_{n-i}([\pi\restriction_n]\phi)) \leftrightarrow (\mathsf{PRDL4})_0([\pi\restriction_i]\phi)$	syntactic equality
5.	$(\mathsf{PRDL4})_{n-i}([\pi\restriction_n]\phi) \rightarrow (\mathsf{PRDL4})_0([\pi\restriction_i]\phi)$	3, 4
6.	$[\pi\restriction_n]\phi \rightarrow [\pi\restriction_i]\phi$	1, 2, 5

Step 4 is justified, because both $(\mathsf{PRDL4})_{n-i}([\pi\restriction_n]\phi)$ and $(\mathsf{PRDL4})_0([\pi\restriction_i]\phi)$ result from the same number of applications of (PRDL4) to $[\pi\restriction_n]\phi$ and $[\pi\restriction_i]\phi$ respectively. The latter two formulas are syntactically equal, except for the restriction parameter. The formulas $(\mathsf{PRDL4})_{n-i}([\pi\restriction_n]\phi)$ and $(\mathsf{PRDL4})_0([\pi\restriction_i]\phi)$ are thus also syntactically equal,[17] except for the restriction parameters, which are $n - i$ or 0 in the first case and 0 in the latter. Setting the restriction parameters of the first formula to 0, will thus give us equivalent formulas. □

Example 7.5 We now consider an agent with two plan revision rules: $\mathsf{Rule} = \{a \rightsquigarrow a; a,\ a; a; a \rightsquigarrow b\}$ and we assume that $[a\restriction_0]\phi$ and $[b\restriction_0]\phi$. We want to prove $\forall n \in \mathbb{N} : [a\restriction_n]\phi$. Along similar lines of reasoning as in Example 7.3, i.e., by using axiom (PRDL4) to rewrite $[a\restriction_n]\phi$, we can conclude that we may again use assumption (7.10) from Example 7.3. We have to prove the following, taking the "last line" of the rewriting of $[a\restriction_{n+1}]\phi$ by (PRDL4).

$$\bigwedge_i [a\restriction_0][(a; (a^i))\restriction_{n-i}]\phi \qquad \text{for } 0 \leq i \leq n \tag{7.12}$$

$$\bigwedge_i [(b; (a^{i-2}))\restriction_{n-i}]\phi \qquad \text{for } 2 \leq i \leq n \tag{7.13}$$

$$[(a; a; (a^n))\restriction_0]\phi \tag{7.14}$$

The formulas (7.12) and (7.14) were proven in the example above, using assumption (7.10). We will prove (7.13) by proving $\bigwedge_i [(a^{i-2})\restriction_{n-i}]\phi$ and using (GEN) to derive the desired formula.

[17] That is, modulo swapping of conjuncts.

In the proof below, let $3 \leq i \leq n$ and let $0 \leq r \leq n$ in the first line and $0 \leq r \leq n-3$ in the second line.

1.	$\bigwedge_r[(a;(a^r))\restriction_{n-r}]\phi$	assumption (7.10)
2.	$\bigwedge_r[(a;(a^r))\restriction_{n-r-3}]\phi$	1, Proposition 7.5
3.	$\bigwedge_i[(a;(a^{i-3}))\restriction_{n-i}]\phi$	where $r = i-3$
4.	$\bigwedge_i[(a^{i-2})\restriction_{n-i}]\phi$	definition of a^i
5.	$\bigwedge_i[b\restriction_0][(a^{i-2})\restriction_{n-i}]\phi$	4, (GEN)
6.	$\bigwedge_i[b\restriction_0][(a^{i-2})\restriction_{n-i}]\phi \leftrightarrow \bigwedge_i[(b;(a^{i-2}))\restriction_{n-i}]\phi$	(PRDL4)
7.	$\bigwedge_i[(b;(a^{i-2}))\restriction_{n-i}]\phi$	5, 6, (MP)

The above proves $[(b;(a^{i-2}))\restriction_{n-i}]\phi$ for $3 \leq i \leq n$. If $i = 2$, we need to prove $[b\restriction_{n-2}]\phi$. According to axiom (PRDL4), this is equivalent to proving $[b\restriction_0]\phi$.[18] This was given, so we are done. △

In Section 7.5.1, we have presented an infinitary axiom system to prove properties of non-restricted 3APL plans. As an infinitary axiom system is difficult to use, we have suggested to use induction on the number of plan revision rule applications, i.e., on the restriction parameter, in an expression. Some examples have been worked out to illustrate this approach. As the examples show, it is doable (at least for the example cases) to use induction on the number of plan revision rule applications. It is however a fairly complicated undertaking. Future research will have to show whether this type of reasoning is amenable to some kind of automation, and what the limits of the approach are.

7.6 Plan Revision Rules versus Procedures

The operational semantics of (parameterless) procedures is similar to that of plan revision rules. The operational semantics of a procedure $p \Leftarrow S$ where p is the procedure name and the statement S is the body of the procedure, can be defined by a transition $\langle p; S', \sigma\rangle \rightarrow \langle S; S', \sigma\rangle$, where S' is a statement. If we compare this semantics to the semantics of plan revision rules of Definition 7.7, we can see that both are so-called body-replacement semantics: if the head of a plan revision rule or the name of a procedure occur at the head of a statement that is to be executed, the head or the procedure name are replaced by the body of the rule or the procedure respectively.

Because of this similarity, one might think that techniques used for reasoning about procedures can be used to reason about plan revision rules. This however turns out not to be the case, due to the non-compositional semantics of the sequential composition operator in 3APL (see introduction of Section 7.3). In this section, we will elaborate on this issue by studying inference rules of Hoare logic for reasoning about procedures (see for example [de Bakker, 1980, Apt, 1981]

[18] By (PRDL4) we have $[b\restriction_{n-2}]\phi \leftrightarrow [b\restriction_0][\epsilon\restriction_{n-2}]\phi$ and by (PRDL3): $[b\restriction_0][\epsilon\restriction_{n-2}]\phi \leftrightarrow [b\restriction_0]\phi$.

for a detailed explanation of Hoare logic). We will also show that reasoning by induction on the number of plan revision rule applications and reasoning about procedures using Hoare logic inference rules, although very different at first sight, actually do have similarities.

7.6.1 Reasoning about Procedures

Hoare logic is used for reasoning about programs. Inference rules are defined to derive so-called Hoare triples. A Hoare triple is of the form $\{\phi_1\}\ S\ \{\phi_2\}$ and intuitively means that if ϕ_1 holds, ϕ_2 will always hold after the execution of the statement S.[19] To reason about *non-recursive* procedures, the following inference rule can be defined for a procedure $p \Leftarrow S$ (for simplicity, we assume we only have one procedure) with procedure name p and body S.

$$\frac{\{\phi_1\}\ S\ \{\phi_2\}}{\{\phi_1\}\ p\ \{\phi_2\}}$$

The rule states that if we can prove that ϕ_2 holds after the execution of the body S of the procedure (assuming ϕ_1 holds before execution), we can infer that ϕ_2 holds after the procedure call p.

If the procedure $p \Leftarrow S$ is *recursive*, that is, if p is called in S, the rule above will still be sound, but a system with only this rule for reasoning about procedure calls will not be complete (see also [Apt, 1981]). An attempt at proving $\{\phi_1\}\ p\ \{\phi_2\}$ results in an infinite regression. The following rule [Apt, 1981], which is a variant of so-called Scott's induction rule (see for example [de Bakker, 1980]), is meant to overcome this difficulty.

Definition 7.21 *(Scott's induction rule)*

$$\frac{\{\phi_1\}\ p\ \{\phi_2\} \vdash \{\phi_1\}\ S\ \{\phi_2\}}{\{\phi_1\}\ p\ \{\phi_2\}}$$

The rule states that if we can prove $\{\phi_1\}\ S\ \{\phi_2\}$ from the assumption that $\{\phi_1\}\ p\ \{\phi_2\}$, we can infer $\{\phi_1\}\ p\ \{\phi_2\}$. Using this rule for reasoning about procedure calls, a complete proof system can be obtained [Apt, 1981].[20]

In a proof of a property of a procedural program, the rule above is (often) used in combination with the following rule for sequential composition.

Definition 7.22 *(rule for sequential composition)*

$$\frac{\{\phi_1\}\ S\ \{\phi_2\} \quad \{\phi_2\}\ S'\ \{\phi_3\}}{\{\phi_1\}\ S; S'\ \{\phi_3\}}$$

[19] The Hoare triple $\{\phi_1\}\ S\ \{\phi_2\}$ can be characterized in dynamic logic by the formula $\phi_1 \rightarrow [S]\phi_2$.

[20] Note that this is a proof rule for deriving partial correctness specifications, a Hoare triple $\{\phi_1\}\ p\ \{\phi_2\}$ meaning that *if* p terminates, ϕ_2 will hold after execution of p (provided that p is executed in a state in which ϕ_1 holds). If p does not terminate, anything is derivable for p. The rule cannot be used to prove termination of p.

Consider for example a procedure $p \Leftarrow p$ and suppose we want to prove $\{\phi_1\}\ p; S\ \{\phi_3\}$ (p is non-terminating, so we should be able to prove this for any ϕ_1 and ϕ_3). We then have to prove $\{\phi_1\}\ p\ \{\phi_2\}$ and $\{\phi_2\}\ S\ \{\phi_3\}$ for some ϕ_2. If we take $\phi_2 = \mathbf{0}$, i.e., falsum, the second conjunct follows immediately. In proving $\{\phi_1\}\ p\ \{\mathbf{0}\}$, which we will refer to as H, we use Scott's induction rule and we thus have to prove H from the assumption that H. This is immediate, concluding the proof.

The point of this example is the following. Using Scott's induction rule, we can prove properties of a procedure call p. If we want to prove a property of a statement involving the sequential composition of this procedure call and some other statement S, we can use properties proven of the procedure call (obtained using Scott's induction rule) and compose it with properties proven of S by means of the rule for sequential composition. In particular, this technique can be applied to for example a procedure $p \Leftarrow p; S$, where an assumption about p can be used to prove properties of $p; S$. Scott's induction rule for proving properties of procedure calls is thus most useful if used in combination with the rule for sequential composition.

Scott's induction rule for plan revision rules

A question one might ask, is whether a variant of Scott's induction rule can be used to reason about plan revision rules. Assuming one plan revision rule $\pi_h \rightsquigarrow \pi_b$, the following rule could be formulated.

$$\frac{\{\phi_1\}\ \pi_h\ \{\phi_2\} \vdash \{\phi_1\}\ \pi_b\ \{\phi_2\}}{\{\phi_1\}\ \pi_h\ \{\phi_2\}}$$

Assume for the moment that it is possible to use this rule to prove $\{\phi_1\}\ \pi_h\ \{\phi_2\}$ for some plan revision rule $\pi_h \rightsquigarrow \pi_b$ and properties ϕ_1 and ϕ_2. The question now is, whether the fact that we can prove $\{\phi_1\}\ \pi_h\ \{\phi_2\}$, will do us any good if we want to prove properties of more complex plans such as $\pi_h; \pi$.

Proving properties of $\pi_h; \pi$ based on properties proven of π_h, would have to be done using the rule for sequential composition. This rule is however not sound in the context of plan revision rules. In general, it is *not* the case that $\mathcal{O}(\pi_1; \pi_2)(\sigma) \subseteq \mathcal{O}(\pi_2)(\mathcal{O}(\pi_1)(\sigma))$ (see also the introduction of Section 7.3). Let $\Sigma_1 = \mathcal{O}(\pi_1)(\sigma)$ and $\Sigma_2 = \mathcal{O}(\pi_2)(\Sigma_1)$. If ϕ_2 holds in all states in Σ_1 (if ϕ_1 holds in σ), then ϕ_3 will hold in all states in Σ_2 by assumption. Let $\Sigma_3 = \mathcal{O}(\pi_1; \pi_2)(\sigma)$ and let $\sigma' \in \Sigma_3$, but $\sigma' \notin \Sigma_2$. Then we may not conclude that ϕ_3 will hold in σ' and therefore the rule is not sound.

The fact that we can prove $\{\phi_1\}\ \pi_h\ \{\phi_2\}$, will thus not help if we want to prove properties of a plan like $\pi_h; \pi$, because we do not have a rule for sequential composition. In particular, the assumption $\{\phi_1\}\ \pi_h\ \{\phi_2\}$ will not help to prove $\{\phi_1\}\ \pi_b\ \{\phi_2\}$, even if $\pi_b = \pi_h; \pi$. It is thus not clear whether it should be possible in the general case to prove $\{\phi_1\}\ \pi_b\ \{\phi_2\}$ from the assumption $\{\phi_1\}\ \pi_h\ \{\phi_2\}$. Moreover, the rule above is not sound for agents with more than one plan

revision rule. It is then in general *not* the case that $\mathcal{O}(\pi_b)(\sigma) = \mathcal{O}(\pi_h)(\sigma)$, rather $\mathcal{O}(\pi_b)(\sigma) \subseteq \mathcal{O}(\pi_h)(\sigma)$. Therefore, we may not conclude $\{\phi_1\}\ \pi_h\ \{\phi_2\}$ from a proof of $\{\phi_1\}\ \pi_b\ \{\phi_2\}$.

7.6.2 Induction

In Section 7.6.1 we argued that, although the operational semantics of plan revision rules and procedure calls are very similar, we cannot use Scott's induction rule, which is used for reasoning about procedure calls, to reason about plan revision rules. Our solution to the issue of reasoning about plan revision rules as presented in this chapter, is to do induction on the number of plan revision rule applications. In this section, we will elaborate on why Scott's induction rule is called an *induction* rule and by doing this, we will see that induction on the number of plan revision rule applications and induction as used in Scott's induction rule, have strong similarities.

At first sight, it does not look like using Scott's induction rule involves doing induction, because we do not see formulas parameterized with natural numbers n and $n+1$. To see why the rule actually *is* an induction rule, we first rephrase the rule of Definition 7.21 and adopt notation used by De Bakker [de Bakker, 1980]. Ω is used to denote a non-terminating statement (similar to the `fail` statement mentioned in the proof of Theorem 7.2). The first element of a tuple $\langle \ldots \mid \ldots \rangle$ is used to indicate the procedures, in the presence of which the formula of the second element should hold.

$$\frac{\{\phi_1\}\ \Omega\ \{\phi_2\} \qquad \langle \mid \{\phi_1\}\ p\ \{\phi_2\} \vdash \{\phi_1\}\ S\ \{\phi_2\}\rangle}{\langle p \Leftarrow S \mid \{\phi_1\}\ p\ \{\phi_2\}\rangle} \tag{7.15}$$

The rule above is an instantiation of a more general version of this rule for multiple procedures [de Bakker, 1980]. The first antecedent is derived from this general rule, but could be omitted in this form: Ω is a non-terminating statement and therefore the triple $\{\phi_1\}\ \Omega\ \{\phi_2\}$ is valid for any ϕ_1, ϕ_2. We will however not eliminate it for the purpose of comparing this rule with reasoning about plan revision rules.

Now, consider a procedure $p \Leftarrow S$ and let S^n be defined as follows: $S^0 = \Omega$ and $S^{n+1} = S[S^n/p]$, where $S[S^n/p]$ means that every occurrence of p in S is replaced by S^n. If for example $S = p; S'$, then $S^1 = S^0; S' = \Omega; S'$, $S^2 = S^1; S' = (\Omega; S'); S'$, etc.

Using this substitution construction, we can define the meaning $\mathcal{M}$ of a procedure $p \Leftarrow S$ in the following way (see Apt [Apt, 1981]): $\mathcal{M}(p) = \bigcup_{n=0}^{\infty} \mathcal{M}(S^n)$. From this, we can conclude that $\langle p \Leftarrow S \mid \{\phi_1\}\ p\ \{\phi_2\}\rangle$ is true iff $\forall n : \langle p \Leftarrow S^n \mid \{\phi_1\}\ p\ \{\phi_2\}\rangle$ is true [Apt, 1981]. Therefore, the induction rule above is equivalent with the following rule.

$$\frac{\{\phi_1\}\ \Omega\ \{\phi_2\} \qquad \langle \mid \{\phi_1\}\ p\ \{\phi_2\} \vdash \{\phi_1\}\ S\ \{\phi_2\}\rangle}{\forall n : \langle p \Leftarrow S^n \mid \{\phi_1\}\ p\ \{\phi_2\}\rangle} \tag{7.16}$$

The meaning of a procedure call p of a procedure $p \Leftarrow S$ is equivalent with the meaning of S. More in general, the meaning of a statement S' in which a call to procedure $p \Leftarrow S$ occurs, is equivalent with the meaning of the statement $S'[S/p]$, i.e., the statement S' in which all occurrences of p are replaced with S (see [de Bakker, 1980]). Therefore, we may replace p with S^n in rule (7.16) and we may replace occurrences of p in S with S^n. We have by definition that $S[S^n/p] = S^{n+1}$, yielding the following equivalent rule[21].

$$\frac{\{\phi_1\}\ \Omega\ \{\phi_2\} \qquad \forall n : (\{\phi_1\}\ S^n\ \{\phi_2\} \vdash \{\phi_1\}\ S^{n+1}\ \{\phi_2\})}{\forall n : \{\phi_1\}\ S^n\ \{\phi_2\}} \tag{7.17}$$

This rule, which is equivalent with Scott's induction rule, demonstrates clearly why Scott's induction rule is called an *induction* rule. The idea of proving properties of a 3APL agent of the form $\forall n : \vdash_{\mathsf{Rule}} [\pi\restriction_n]\phi$ by induction on n, is that we prove $[\pi\restriction_0]\phi$ and $\forall n : ([\pi\restriction_n]\phi \vdash_{\mathsf{Rule}} [\pi\restriction_{n+1}]\phi)$. The similarity between the two approaches is thus that induction on respectively the number of procedure calls and plan revision rule applications is done (implicitly or explicitly).

The important difference however is that the statement S in rule (7.17) corresponds with the body of a procedure p in the equivalent rule (7.15). The plan π on the other hand does not correspond with the body of a plan revision rule, but rather refers to the initial plan of the agent. Related to this is the fact that rule (7.17) or the equivalent rule (7.15) can be used in combination with the rule for sequential composition, as explained in Section 7.6.1. In the case of using induction to reason about 3APL plans, this is impossible.

Concluding, the general idea of doing induction on the number of plan revision rule applications is less obscure than one might have thought at first sight, because of the similarity with the standard Scott's induction rule. The way in which induction can be used to prove properties of plans or programs, however differs between the two approaches due to the non-compositional semantics of the sequential composition operator in plans, as a result of the presence of plan revision rules.

7.7 Conclusion

In this chapter, we presented a dynamic logic for reasoning about 3APL agents, tailored to handle the plan revision aspect of the language. As we argued, 3APL plans cannot be analyzed by structural induction, which means that standard propositional dynamic logic cannot be used to reason about 3APL plans. Instead, we proposed a logic of restricted plans with sound and complete axiomatization. We also showed that this logic can be extended to a logic for non-restricted plans. This however results in an infinitary axiom system. We

[21] We omit the procedure declaration $p \Leftarrow S$, because there are no occurrences of p in either S^n or S^{n+1} by definition.

suggested that a possible way of dealing with the infinitary nature of the axiom system, is reasoning by induction on the restriction parameter. We showed some examples of how this could be done. Finally, we discussed the relation between plan revision rules and procedures. In particular, we argued that there is a similarity between the use of Scott's induction rule for reasoning about procedures, and the use of induction on the number of plan revision rules applications for reasoning about plan revision rules.

Concluding, being able to do structural induction is usually considered an essential property of programs in order to reason about them. As 3APL plans lack this property, it is not at all obvious that it should be possible to reason about them, especially using a clean logic with sound and complete axiomatization. The fact that we succeeded in providing such a logic, thus at least demonstrates this possibility. The resulting infinitary axiom system is nevertheless more of theoretical than practical importance. Future research will have to show whether reasoning by doing induction on the number of plan revision rule applications is amenable to some kind of automation, working towards an extension of these results to a more practical setting. Another important line of research is to find interesting subclasses of plan revision rules that *can* be analyzed by structural induction. One such class is defined in Chapter 8.

Chapter 8

Compositional Semantics of Plan Revision

This chapter is based on [van Riemsdijk and Meyer, 2006]. The main issue which arises with the introduction of plan revision rules, is the issue of *compositionality* of semantics of plans. Due to the introduction of plan revision rules, the semantics of plans is not compositional, which gives rise to problems when trying to reason about 3APL programs. A proof system for a programming language will typically contain rules by means of which properties of the entire program can be proven by proving properties of the parts of which the program is composed. Since the semantics of 3APL plans is not compositional, this is problematic in the case of 3APL. One way of trying to approach this problem is by defining a specialized logic for 3APL which tries to circumvent the issue, as was done in Chapter 7. The resulting logic, however, is non-standard and can be difficult to use, due to the infinitary axiom system.

The approach we take in this chapter, is to try to *restrict* the allowed plan revision rules, such that the semantics of plans becomes compositional in some sense. It is not immediately obvious what kind of restriction would yield the desired result. In this chapter, we propose such a restriction and prove that the semantics of plans in that case is compositional.

The outline of the chapter is as follows. In Section 8.1, we present the syntax and semantics of a simplified version of 3APL. In Section 8.2 we elaborate on the issue of compositionality and explain why the semantics of full 3APL is not compositional. In Section 8.3 we present our proposal for a restricted version of plan revision rules, and prove that the semantics of plans is compositional, given this restriction on plan revision rules.

8.1 3APL

8.1.1 Syntax

The 3APL language we use in this chapter, is almost the same as the language of Chapter 7. The only difference is that we omit abstract plans from the language of plans, for reasons of simplicity. This is no fundamental restriction, since abstract plans can be modeled as non-executable basic actions, i.e., actions for which the function $\mathcal{T}$ is always undefined. We repeat the definitions of Chapter 7 here, for ease of reference.

Definition 8.1 *(belief bases)* Assume a propositional language $\mathcal{L}$ with typical formula p and the connectives $\wedge$ and $\neg$ with the usual meaning. Then the set of belief bases Σ with typical element σ is defined to be $\wp(\mathcal{L})$.

Definition 8.2 *(plans)* Assume that a set $\mathsf{BasicAction}$ with typical element a is given. The set of plans Plan with typical element π is then defined as follows.

$$\pi ::= a \mid \pi_1; \pi_2$$

We use ϵ to denote the empty plan and identify $\epsilon; \pi$ and $\pi; \epsilon$ with π.

Definition 8.3 *(plan revision rules)* The set of plan revision rules $\mathcal{R}$ is defined as follows: $\mathcal{R} = \{\pi_h \rightsquigarrow \pi_b \mid \pi_h, \pi_b \in \mathsf{Plan}, \pi_h \neq \epsilon\}$.

Definition 8.4 *(3APL agent)* A 3APL agent $\mathcal{A}$ is a tuple $\langle \mathsf{PR}, \mathcal{T} \rangle$ where $\mathsf{PR} \subseteq \mathcal{R}$ is a finite set of plan revision rules and $\mathcal{T} : (\mathsf{BasicAction} \times \Sigma) \rightarrow \Sigma$ is a partial function, expressing how belief bases are updated through basic action execution.

Definition 8.5 *(configuration)* Let Σ be the set of belief bases and let Plan be the set of plans. Then $\mathsf{Plan} \times \Sigma$ is the set of configurations of a 3APL agent.

8.1.2 Semantics

The semantics of 3APL is again as in Chapter 7. We repeat the definitions here.

Definition 8.6 *(action execution)* Let $a \in \mathsf{BasicAction}$.

$$\frac{\mathcal{T}(a, \sigma) = \sigma'}{\langle a; \pi, \sigma \rangle \rightarrow_{exec} \langle \pi, \sigma' \rangle}$$

Definition 8.7 *(rule application)* Let $\rho : \pi_h \rightsquigarrow \pi_b \in \mathsf{PR}$.

$$\langle \pi_h \bullet \pi, \sigma \rangle \rightarrow_{apply} \langle \pi_b \bullet \pi, \sigma \rangle$$

Definition 8.8 *(computation sequences)* The set Σ^+ of finite computation sequences is defined as $\{\sigma_1, \ldots, \sigma_i, \ldots, \sigma_n \mid \sigma_i \in \Sigma, 1 \leq i \leq n, n \in \mathbb{N}\}$.

Definition 8.9 *(function for calculating computation sequences)* Let $x_i \in \{exec, apply\}$ for $1 \leq i \leq m$. The function $\mathcal{C}^{\mathcal{A}} : (\mathsf{Plan} \times \Sigma) \rightarrow \wp(\Sigma^+)$ is then as defined below.

$$\mathcal{C}^{\mathcal{A}}(\pi, \sigma) = \{\sigma, \ldots, \sigma_m \in \Sigma^+ \mid \langle \pi, \sigma \rangle \rightarrow_{x_1} \ldots \rightarrow_{x_m} \langle \epsilon, \sigma_m \rangle \text{ is a finite sequence of transitions in } \mathsf{Trans}_{\mathcal{A}}\}.$$

Definition 8.10 *(operational semantics)* Let $\kappa : \Sigma^+ \rightarrow \Sigma$ be a function yielding the last element of a finite computation sequence, extended to handle sets of computation sequences as follows, where I is some set of indices: $\kappa(\{\delta_i \mid i \in I\}) = \{\kappa(\delta_i) \mid i \in I\}$. The operational semantic function $\mathcal{O}^{\mathcal{A}} : \mathsf{Plan} \rightarrow (\Sigma \rightarrow \wp(\Sigma))$ is defined as follows:

$$\mathcal{O}^{\mathcal{A}}(\pi)(\sigma) = \kappa(\mathcal{C}^{\mathcal{A}}(\pi, \sigma)).$$

We will in the sequel omit the superscript $\mathcal{A}$ to functions as defined above, for reasons of presentation.

8.2 3APL and Non-Compositionality

In this section, we revisit the issue of compositionality in the context of 3APL. Before we go into discussing why the semantics of 3APL plans is not compositional, we consider compositionality of standard procedural languages.

8.2.1 Compositionality of Procedural Languages

The semantics of standard procedural languages such as described in [de Bakker, 1980, Chapter 5] are compositional. Informally, a semantics for a programming language is compositional if the semantics of a composed program can be defined in terms of the semantics of the parts of which it is composed. To be more specific, the meaning of a composed program $S_1; S_2$ should be definable in terms of the meaning of S_1 and S_2, for the semantics to be compositional.

A semantics can be defined directly in a compositional way, in which case the semantics is often termed a denotational semantics [de Bakker, 1980]. Alternatively, a semantics can be *defined* in a *non*-compositional way, such as an operational semantics defined using computation sequences, while it still *satisfies a compositionality property*. In this chapter, we focus on the latter case. It turns out that the operational semantics for a procedural language such as discussed in [de Bakker, 1980, Chapter 5] satisfies such a compositionality property, while the operational semantics of 3APL of Definition 8.10 does not. All results and definitions with respect to procedural languages which we refer to in Section 8.2, can be found in [de Bakker, 1980, Chapter 5].

An operational semantics of a procedural language can be defined analogously to the operational semantics of 3APL of Definition 8.10, where plans are

statements and belief bases are states (see also Section 7.6). Both operational semantics are defined in a non-compositional way, since they do not use the structure of the plan or statement to define its semantics. Nevertheless, the operational semantics of a procedural language does satisfy a compositionality property, i.e., the following holds: $\mathcal{O}(S_1; S_2)(\sigma) = \mathcal{O}(S_2)(\mathcal{O}(S_1)(\sigma))$, where S_1 and S_2 are statements. This property specifies that the set of states possibly resulting from the execution of a composed statement $S_1; S_2$ in σ is equal to the set of states resulting from the execution of S_2 in all states resulting from the execution of S_1 in σ.

8.2.2 Non-Compositionality of 3APL

While the presented compositionality property is termed "natural" in [de Bakker, 1980, Chapter 5], it is *not* satisfied by the operational semantics of 3APL, i.e., it is not the case that $\mathcal{O}(\pi_1; \pi_2)(\sigma) = \mathcal{O}(\pi_2)(\mathcal{O}(\pi_1)(\sigma))$ always holds. The reason for this lies in the presence of plan revision rules. Take for example an agent with one plan revision rule $a; b \rightsquigarrow c$. Let σ_{ab} and σ_c be the belief bases resulting from the execution of actions a followed by b, and c in σ, respectively. We then have that $\mathcal{O}(a; b)(\sigma) = \{\sigma_{ab}, \sigma_c\}$, i.e., the agent can either execute the actions a and b one after the other, or it can apply the plan revision rule and then execute c.

If the semantics of 3APL plans would have been compositional, we would also have that $\mathcal{O}(b)(\mathcal{O}(a)(\sigma)) = \{\sigma_{ab}, \sigma_c\}$. This is however not the case, since $\mathcal{O}(b)(\mathcal{O}(a)(\sigma)) = \{\sigma_{ab}\}$.[1] This stems from the fact that if one "breaks" the composed plan $a; b$ in two, one can no longer apply the plan revision rule $a; b \rightsquigarrow c$, because this rule can only be applied if the composed plan $a; b$ is considered. The set of belief bases $\mathcal{O}(a)(\sigma)$ only contains those resulting from the execution of a. The action b is then executed on those belief bases, yielding $\mathcal{O}(b)(\mathcal{O}(a)(\sigma))$. The result thus does not contain σ_c.

8.2.3 Reasoning about 3APL

This non-compositionality property of 3APL plans gives rise to problems when trying to define a proof system for reasoning about 3APL plans. In standard procedural languages, the following proof rule is part of any Hoare logic for such a language [de Bakker, 1980], where p, p' and q are assertions.

$$\frac{\{p\}\ S_1\ \{p'\} \qquad \{p'\}\ S_2\ \{q\}}{\{p\}\ S_1; S_2\ \{q\}} \tag{8.1}$$

This rule specifies that one can reason about a composed program by proving properties of the parts of which it is composed. The soundness of this rule depends on the fact that $\mathcal{O}(S_1; S_2)(\sigma) = \mathcal{O}(S_2)(\mathcal{O}(S_1)(\sigma))$. Because this property

[1] Note that $\mathcal{O}(b)(\mathcal{O}(a)(\sigma)) \subseteq \mathcal{O}(a; b)(\sigma)$.

does not hold for 3APL plans, a similar rule for 3APL would not be sound (see also the discussion in Section 7.6). Nevertheless, one would still want to reason about composed 3APL plans.

In Chapter 7, we have presented a specialized dynamic logic for this purpose. While that chapter aims at reasoning about full 3APL, we take a different approach in this chapter. Here, we investigate whether we can somehow *restrict* plan revision rules, such that the semantics of plans becomes compositional (in some sense). The idea is that given such a compositional semantics, it will be possible to come up with a more standard and easy to use proof system for 3APL.

8.3 Compositional 3APL

One obvious candidate for a restricted version of plan revision rules is the restriction to rules with an atomic head, i.e., to rules of the form $a \rightsquigarrow \pi$. These rules are very similar to procedures, apart from the fact that an action a could either be transformed using a plan revision rule, *or* executed directly. In contrast with actions, procedure variables cannot be executed, i.e., they can only be replaced by the body of a procedure. It is easy to see that a semantics for 3APL with only these plan revision rules would be compositional.

However, this kind of plan revision rules would capture very little of the general plan revision capabilities of the non-restricted rules. The challenge is thus to find a less restrictive kind of plan revision rules, which would still satisfy the desired compositionality property. Finding such a restricted kind of plan revision rules is non-trivial. We discuss the line of reasoning by which it can be obtained in Section 8.3.1. In Section 8.3.2, we present and explain the theorem that expresses that the proposed restriction on plan revision rules indeed establishes (some form of) compositionality. Finally, in Section 8.3.3, we briefly address the issue of reasoning about 3APL with restricted plan revision rules, and point to directions for future research regarding this issue.

8.3.1 Restricted Plan Revision Rules

The restriction to plan revision rules that we propose is given in Definition 8.11 below, and can be understood by trying to get to the essence of the compositionality problem arising from non-restricted plan revision rules.

First, we have to observe that the general kind of compositionality as specified in Section 8.2.1 for procedural languages is in general not obtainable for 3APL, if the set of plan revision rules contains a rule with a non-atomic head. The property specifies that the semantics of a composed plan (or program) should be definable in terms of the parts of which it is composed. The property however does not specify how a composed plan should be broken down into parts. That is, for a plan to be compositional in the general sense, com-

positionality should hold, no matter how the plan is decomposed. Consider for example the plan $a; b; c$. It should then be the case that $\mathcal{O}(a; b; c)(\sigma) = \mathcal{O}(c)(\mathcal{O}(a; b)(\sigma)) = \mathcal{O}(b; c)(\mathcal{O}(a)(\sigma))$, i.e., the compositionality property should hold, no matter whether the plan is decomposed into $a; b$ and c, or a and $b; c$.

If a set of plan revision rules however contains a rule with a non-atomic head, it is always possible to come up with a plan (and belief base and belief update function) for which this property does not hold. This plan should contain the head of the plan revision rule. If the decomposition of the plan is then chosen such that it "breaks" this occurrence of the head of the rule in the plan, the compositionality property in general does not hold for this decomposition. This is because the plan revision rule can in that case not be applied when calculating the result of the operational semantic function. Consider for example the plan revision rule $a; b \rightsquigarrow c$ and the plan $a; b; c$. If the plan is decomposed into a and $b; c$, the rule cannot be applied and thus $\mathcal{O}(a; b; c)(\sigma) = \mathcal{O}(b; c)(\mathcal{O}(a)(\sigma))$ does not always hold.

The question is now which kind of compositionality *can* be obtained for 3APL. We have established that being allowed to decompose a composed plan into arbitrary parts for a definition of compositionality gives rise to problems in the case of 3APL. That is, the standard definition of compositionality will always be problematic if we want to consider plan revision rules with a non-atomic head. Since we want our restriction on plan revision rules to allow at least some form of non-atomicity (because otherwise we would essentially be considering procedures), we have to come up with another definition of compositionality if we want to make any progress.

The idea that we propose is essentially to take the operational meaning of a plan as the basis for a compositionality property. When executing a plan π, either the first action of π is executed, or an applicable plan revision rule is applied. In the first case, π has to be of the form $a; \pi_r$[2], and in the latter case of the form $\pi_h; \pi'_r$, given an applicable plan revision rule of the form $\pi_h \rightsquigarrow \pi_b$. Taking this into account, we are, broadly speaking, looking for a restriction to plan revision rules which allows us to decompose π into a and π_r, or π_h and π'_r. To be more specific, it should be possible to execute a and then consider π_r separately, or to apply the specified plan revision rule and then consider the body of the rule π_b and the rest of the plan, i.e., π'_r, separately. That is, we are after something like the following compositionality property:[3]

$$\mathcal{O}(\pi)(\sigma) = \mathcal{O}(\pi_r)(\mathcal{O}(a)(\sigma)) \cup \mathcal{O}(\pi'_r)(\mathcal{O}(\pi_b)(\sigma)). \tag{8.2}$$

In order to come up with a restriction on plan revision rules that gives us such a property, we have to understand why this property does not always hold in the presence of non-restricted plan revision rules. Essentially, what this property

[2]The subscript r here indicates that π_r is the *rest* of the plan π.

[3]The property that will be proven in Section 8.3.2 differs slightly, as it takes into account the existence of multiple applicable plan revision rules.

specifies is that we can separate the semantics of certain prefixes of the plan π (i.e., a and π_h), from the semantics of the rest of π.

A case in which this is *not* possible, is the following. Consider a plan of the form $\pi_h;\pi'_h;\pi$, and plan revision rules of the form $\pi_h \rightsquigarrow \pi_b$ and $\pi_b;\pi'_h \rightsquigarrow \pi'_b$. We can apply the first rule to this plan, yielding $\pi_b;\pi'_h;\pi$. If the semantics of the plan would be compositional in the sense of (8.2), it should now be possible to consider the semantics of $\pi'_h;\pi$, i.e., the "rest" of the plan, separately. Given the second plan revision rule however, this is not possible: if we separate $\pi_b;\pi'_h;\pi$ into π_b and $\pi'_h;\pi$, we can no longer apply the second plan revision rule, whereas we *can* apply the rule if the plan is considered in its composed form. The semantics of the plan $\pi_h;\pi'_h;\pi$ is thus not compositional, given the two plan revision rules.

This argument is similar to the explanation of why the general notion of compositionality does not hold for 3APL. Contrary to the general case however, we can in the case of compositionality as defined in (8.2), specify a restriction to plan revision rules that prevents this problem from occurring. The restriction will thus allow us to consider the semantics of π'_r (see (8.2)) separately from the semantics of π_b, thereby establishing compositionality Property (8.2).

As explained, if there is a plan revision rule of the form $\pi_h \rightsquigarrow \pi_b$, a plan revision rule with a head of the form $\pi_b;\pi'_h$ is problematic. A restriction one could thus consider, is the restriction that if there is a rule of the form $\pi_h \rightsquigarrow \pi_b$, there should not also be a rule of the form $\pi_b;\pi'_h \rightsquigarrow \pi'_b$, i.e., the body of a rule cannot be equal to the prefix of the head of another rule. This restriction however does not do the trick completely. The reason has to do with the fact that actions from a plan of the form $\pi_b;\pi'_h$ can be executed.

Consider for example a plan $a_1;a_2;b_1;b_2$ and plan revision rules $a_1;a_2 \rightsquigarrow c_1;c_2$ and $c_2;b_1 \rightsquigarrow c_3$. The head of the second rule does not have the form $c_1;c_2;\pi$, i.e., the body of the first rule is not equal to the prefix of the head of another rule. Therefore, according to the suggested restriction, this rule is allowed. We can apply the first rule to the plan, yielding $c_1;c_2;b_1;b_2$. If the compositionality property holds, we should now be able to consider the semantics of $b_1;b_2$ separately. Suppose the action c_1 is executed, resulting in the plan $c_2;b_1;b_2$. Considering the second plan revision rule, we observe that this rule is applicable to this plan. This is however only the case if we consider this plan in its composed form. If we separate the semantics of $b_1;b_2$ as specified by the compositionality Property (8.2), we cannot apply the rule. Given the plan $a_1;a_2;b_1;b_2$ and the two plan revision rules, the compositionality property thus does not hold.

The solution to this problem is to adapt the suggested restriction which considers the body of a rule in relation with the prefix of the head of another rule, to a restriction which consider the *suffix* of the body of a rule in relation with the prefix of the head of another rule. The restriction should thus specify that the suffix of the body of a rule cannot be equal to the prefix of the head of another rule. Under that restriction, the second rule of the example discussed

above would not be allowed, and the compositionality Property (8.2) would hold. This restriction on plan revision rules is specified formally below. The fact that under this restriction, the Property (8.2) (or a slight variation thereof) holds, is formally shown in Section 8.3.2.

Definition 8.11 *(restricted plan revision rules)* Let PR be a set of plan revision rules. Let suff be a function taking a plan and yielding all its suffixes, and let pref be a function taking a plan and yielding all its strict prefixes.[4] We say that PR is *restricted* iff the following holds:

$$\forall \rho \in \mathsf{PR} : (\rho : \pi_h \rightsquigarrow \pi_b) : \neg \exists \rho' \in \mathsf{PR} : (\rho' : \pi'_h \rightsquigarrow \pi'_b) : \big(\mathsf{suff}(\pi_b) \cap \mathsf{pref}(\pi'_h)\big) \neq \emptyset.$$

The fact that we define pref as yielding *strict* prefixes allows the suffix of the body of a plan revision rule to be exactly equal to the head of another rule. This does not violate the compositionality property, and it results in restricted plan revision rules being a superset of rules with an atomic head. Otherwise, a rule $b \rightsquigarrow c$, for example, would not be allowed if there is also a rule $a \rightsquigarrow a; b$, since b, i.e., the suffix of the latter rule, would then by definition be equal to the prefix of the head of the first rule.

8.3.2 Compositionality Theorem

The theorem expressing the compositionality property that holds for plans under a restricted set of plan revision rules, is given below. It is similar to Property (8.2) specified in Section 8.3.1, except that we take into account the existence of multiple applicable plan revision rules. A plan π can thus be decomposed into a and π_r (where π is of the form $a; \pi$), or into π_h^ρ and π_r^ρ (where π is of the form $\pi_h^\rho; \pi_r^\rho$) for any applicable plan revision rule ρ of the form $\pi_h^\rho \rightsquigarrow \pi_b^\rho$.

Theorem 8.1 *(compositionality of semantics of plans)* Let $\mathcal{A}$ be an agent with a restricted set of plan revision rules PR. Let ρ range over the set of rules from PR that are applicable to the plan π, and let π be of the form $\pi_h^\rho; \pi_r^\rho$ for an applicable rule ρ of the form $\pi_h^\rho \rightsquigarrow \pi_b^\rho$. Further, let a be the first action of π, i.e., let π be of the form $a; \pi_r$. We then have for all $\pi \neq \epsilon$ and σ:

$$\mathcal{O}(\pi)(\sigma) \quad = \quad \mathcal{O}(\pi_r)(\mathcal{O}(a)(\sigma)) \ \cup \ \bigcup_\rho \mathcal{O}(\pi_r^\rho)(\mathcal{O}(\pi_b^\rho)(\sigma)).$$

In order to prove this theorem, we use Lemma 8.1 below. This lemma, broadly speaking, specifies that for a plan of the form $\pi_h; \pi$, the following is the case: after application of a plan revision rule of the form $\pi_h \rightsquigarrow \pi_b$, yielding the plan $\pi_b; \pi$, it will always be the case that π_b is executed entirely, before π is executed. Because of this, the semantics of π_b and of π can be considered separately, which is the core of our compositionality theorem.

[4] The plan a is for example a strict prefix of $a; b$, but the plan $a; b$ is not. Further, the empty plan ϵ should not be returned as a prefix, nor as a suffix.

Lemma 8.1 Let $\mathcal{A}$ be an agent with a restricted set of plan revision rules PR, and let $\pi_h \rightsquigarrow \pi_b \in \mathsf{PR}$. We then have that any transition sequence $\langle \pi_b; \pi, \sigma \rangle \rightarrow \ldots \rightarrow \langle \epsilon, \sigma' \rangle$ has the form[5]

$$\langle \pi_b; \pi, \sigma \rangle \rightarrow \ldots \rightarrow \langle \pi, \sigma'' \rangle \rightarrow \ldots \rightarrow \langle \epsilon, \sigma' \rangle$$

such that each configuration in the first part of the sequence, i.e., in $\langle \pi_b; \pi, \sigma \rangle \rightarrow \ldots \rightarrow \langle \pi, \sigma'' \rangle = \theta$, has the form $\langle \pi_i; \pi, \sigma_i \rangle$. That is, π is always the suffix of the plan of the agent in each configuration of θ.

In the proof of this lemma, we use the notion of a plan π' *being suffix* in π with respect to some set of plan revision rules. A plan π' is suffix in π, if π is the suffix of π', i.e., if π' is of the form $\pi_{pre}; \pi$. Further, π_{pre} should be a concatenation of suffixes of the bodies of the relevant set of plan revision rules.

Definition 8.12 *(suffix in π)* Let PR be a set of plan revision rules. Let suf_i with $1 \leq i \leq n$ denote plans that are equal to the suffix of the body of a rule in PR, i.e., for each suf_i there is a rule in PR of the form $\pi_h \rightsquigarrow \pi_r; suf_i$. We say that *a plan π' is suffix in π* with respect to PR, iff π' is of the form $suf_1; \ldots; suf_n; \pi$, and the length of $suf_1; \ldots; suf_n$ is greater than 0.

The idea is that, given a plan of the form $\pi_b; \pi$ which is suffix in π by definition[6], this property is preserved until the plan is of the form π. If this is the case, we have that π is always the (strict) suffix of the plan of each configuration, until the plan equals π. We thus use the preservation of this property to prove Lemma 8.1 (see below).

We need the fact that the part of the plan occurring before π is a sequence of suffixes, in order to prove that π is preserved as the suffix of the plan.[7] The reason is, that if this is the case, we know by the fact that our plan revision rules are restricted, that there cannot occur a rule application which transforms π, thereby violating our requirement that π remains the suffix of the plan of the agent, until the plan becomes equal to π. If a plan is of the form $suf_1; \ldots; suf_n; \pi$, where each suf_i denotes a plan that is equal to the suffix of the body of a plan revision rule, we know that any plan revision rule will only modify a prefix of suf_1, because the plan revision rules are restricted. There cannot be a rule with a head of the form $suf_1; \pi_h$, because this would violate the requirement of restricted plan revision rules.

Proof of Lemma 8.1: Let $\mathcal{A}$ be an agent with a restricted set of plan revision rules PR. Let $\langle \pi_1, \sigma \rangle \rightarrow \langle \pi_2, \sigma' \rangle$ be a transition of $\mathcal{A}$. First, we show that if π_1 is suffix in π (with respect to PR), it has to be the case that π_2 is suffix in π, or that $\pi_2 = \pi$.

[5] In this lemma we omit the labels of transitions, for reasons of presentation.

[6] That is, if π_b is the body of a plan revision rule.

[7] Note that we use the term suffix to refer to suffixes of the plans of the bodies of plan revision rules, and to refer to the suffix of the plan in a configuration.

Assume that π_1 is suffix in π, i.e., let $\pi_1 = suf_1; \ldots; suf_n; \pi$. If $\pi = \epsilon$, the result is immediate. Otherwise, the proof is as follows. A transition from $\langle \pi_1, \sigma \rangle$ results either from the execution of an action, or from the application of an applicable rule.

Let $suf_1 = a; suf'_1$. If action a is executed, π_2 is of the form $suf'_1; \ldots; suf_n; \pi$. If $suf'_1, \ldots, suf_n$ are ϵ, we have that $\pi_2 = \pi$. Otherwise, we have that π_2 is suffix in π.

Let $\rho : \pi_h \rightsquigarrow \pi_b$ be a rule from PR that is applicable to π_1. Then it must be the case that π_1 is of the form $\pi_h; \pi_r$. By the fact that PR is restricted, we have that there is not a rule ρ' of the form $suf_1; \pi' \rightsquigarrow \pi'_b$, i.e., such that suf_1, which is the suffix of the body of a rule, is the prefix of the head of ρ'. Given that ρ is applicable to π_1, it must thus be the case that π_h is a prefix of suf_1, i.e., that suf_1 is of the form $\pi_h; \pi''$. Applying ρ to π_1 thus yields a plan of the form $\pi_b; \pi''; suf_2; \ldots; suf_n; \pi$. Since both π_b and π'' are suffixes of the bodies of rules in PR, we have that π_2 is suffix in π.

We have to show that any transition sequence θ of the form $\langle \pi_b; \pi, \sigma \rangle \rightarrow \ldots \rightarrow \langle \epsilon, \sigma' \rangle$ has a prefix θ' such that π is always a suffix of the plan in each configuration of θ'. Let π_2 be the plan of the second configuration of θ. We have that $\pi_b; \pi$ is suffix in π. Therefore, it must be the case that π_2 is also suffix in π, or that $\pi_2 = \pi$. In the latter case, we have the desired result. In the former case, we have that π is a suffix of π_2, in which case the first two configuration may form a prefix of θ'. Let π_3 be the plan of the third configuration of θ. If π_2 is suffix in π, it has to be the case that π_3 is suffix in π, or that $\pi_3 = \pi$. In the latter case, we are done. In the former case, the first three configurations may form a prefix of θ'. This line of reasoning can be continued. Since θ is a finite sequence, it has to be the case that at some point a configuration of the form $\langle \pi, \sigma'' \rangle$ is reached. This yields the desired result. □

Proof of Theorem 8.1: We have to show the following:

$$\sigma' \in \mathcal{O}(\pi)(\sigma) \quad \Leftrightarrow \quad \sigma' \in \mathcal{O}(\pi_r)(\mathcal{O}(a)(\sigma)) \ \cup \ \bigcup_{\rho} \mathcal{O}(\pi_r^{\rho})(\mathcal{O}(\pi_b^{\rho})(\sigma)).$$

($\Leftarrow$) Follows in a straightforward way from the definitions.
($\Rightarrow$) Let n be the number of plan revision rules applicable to π, where $\pi_h^{\rho_i}$ and $\pi_b^{\rho_i}$ respectively denote the head and body of rule ρ_i. We then have to show:

$$\begin{aligned} \sigma' \in \mathcal{O}(\pi)(\sigma) \quad \Rightarrow \quad & \sigma' \in \mathcal{O}(\pi_r^{\rho_1})(\mathcal{O}(\pi_b^{\rho_1})(\sigma)) \text{ or} \\ & \vdots \\ & \sigma' \in \mathcal{O}(\pi_r^{\rho_n})(\mathcal{O}(\pi_b^{\rho_n})(\sigma)) \text{ or} \\ & \sigma' \in \mathcal{O}(\pi_r)(\mathcal{O}(a)(\sigma)). \end{aligned}$$

If $\sigma' \in \mathcal{O}(\pi)(\sigma)$, then there is a transition sequence of the form

$$\langle \pi, \sigma \rangle \rightarrow_x \ldots \rightarrow_x \langle \epsilon, \sigma' \rangle$$

i.e., if $\pi_h^\rho \rightsquigarrow \pi_b^\rho$ is an arbitrary rule ρ that is applicable to π, where $\pi = \pi_h^\rho; \pi_r^\rho$, there are transition sequences of the form

$$\langle \pi_h^\rho; \pi_r^\rho, \sigma \rangle \rightarrow_{apply} \langle \pi_b^\rho; \pi_r^\rho, \sigma \rangle \rightarrow_x \ldots \rightarrow_x \langle \epsilon, \sigma' \rangle \tag{8.3}$$

or, if $\pi = a; \pi_r$, of the form

$$\langle a; \pi_r, \sigma \rangle \rightarrow_{exec} \langle \pi_r, \sigma'' \rangle \rightarrow_x \ldots \rightarrow_x \langle \epsilon, \sigma' \rangle. \tag{8.4}$$

In case σ' has resulted from a transition sequence of form (8.3), we prove

$$\sigma' \in \mathcal{O}(\pi_r^\rho)(\mathcal{O}(\pi_b^\rho)(\sigma)). \tag{8.5}$$

In case σ' has resulted from a transition sequence of form (8.4), we prove

$$\sigma' \in \mathcal{O}(\pi_r)(\mathcal{O}(a)(\sigma)). \tag{8.6}$$

Assume σ' has resulted from a transition sequence of form (8.3). We then have to prove (8.5), i.e., we have to prove that there is a belief base $\sigma'' \in \mathcal{O}(\pi_b^\rho)(\sigma)$, such that $\sigma' \in \mathcal{O}(\pi_r^\rho)(\sigma'')$. That is, we have to prove that there are transition sequences of the form $\langle \pi_b^\rho, \sigma \rangle \rightarrow \ldots \rightarrow \langle \epsilon, \sigma'' \rangle$, and of the form $\langle \pi_r^\rho, \sigma'' \rangle \rightarrow \ldots \rightarrow \langle \epsilon, \sigma' \rangle$.

By Definitions 8.6 and 8.7, we have that if $\langle \pi_1; \pi_2, \sigma \rangle \rightarrow \langle \pi_1'; \pi_2, \sigma' \rangle$ is a transition for arbitrary plans π_1 and π_2, then $\langle \pi_1, \sigma \rangle \rightarrow \langle \pi_1', \sigma' \rangle$ is also a transition. By Lemma 8.1, we have that there is a prefix of (8.3) of the form $\langle \pi_h^\rho; \pi_r^\rho, \sigma \rangle \rightarrow \langle \pi_b^\rho; \pi_r^\rho, \sigma \rangle \rightarrow \ldots \rightarrow \langle \pi_r^\rho, \sigma'' \rangle$, such that the plan of each configuration in this sequence is of the form $\pi_i; \pi$. From this we can conclude the desired result, i.e., that there are transition sequences of the form $\langle \pi_b^\rho, \sigma \rangle \rightarrow \ldots \rightarrow \langle \epsilon, \sigma'' \rangle$, and of the form $\langle \pi_r^\rho, \sigma'' \rangle \rightarrow \ldots \rightarrow \langle \epsilon, \sigma' \rangle$.

Assume σ' has resulted from a transition sequence of form (8.4). Then proving (8.6) is analogous to proving (8.5), except that we do not need Lemma 8.1. □

8.3.3 Reasoning about Compositional 3APL

As argued, an important reason for defining a variant of 3APL with a compositional semantics, is that it is more likely that it will be possible to come up with a more standard and easy to use proof system for such a language. A natural starting point for such an effort is the definition of a proof rule for sequential composition, analogous to rule (8.1), as specified below (we use the notation of Theorem 8.1).

$$\frac{\{p\}\ a\ \{p'\} \quad \{p'\}\ \pi_r\ \{q\} \qquad \bigwedge_\rho \big(\{p\}\ \pi_b^\rho\ \{p'\} \text{ and } \{p'\}\ \pi_r^\rho\ \{q\}\big)}{\{p\}\ \pi\ \{q\}} \tag{8.7}$$

The soundness proof of this rule is analogous to the soundness proof of rule (8.1) [de Bakker, 1980, Chapter 2], but using Theorem 8.1 instead of $\mathcal{O}(S_1; S_2)(\sigma) = \mathcal{O}(S_2)(\mathcal{O}(S_1)(\sigma))$. A complete proof system for compositional 3APL would however also need an induction rule. We conjecture that it will be possible to define an analogue of Scott's induction rule [de Bakker, 1980, Chapter 5] which is used for proving properties of recursive procedures, for reasoning about plans in the context of plan revision rules. Investigating this is however left for future research.

PART III

SOFTWARE ENGINEERING ASPECTS

Chapter 9

Goal-Oriented Modularity

This chapter is based on [van Riemsdijk et al., 2006a]. *Modularization* is widely recognized as a central issue in software engineering [Meyer, 1988] [Ghezzi et al., 1991, Bergstra et al., 1990]. A system which is composed of modules, i.e., relatively independent units of functionality, is called modular. A programming language which adheres to this principle of modularization supports the decomposition of a system into modules. The principle lies, e.g., at the basis of procedural programming and object-oriented programming, in which respectively (libraries of) procedures and classes form the modules.

An important advantage of modularity in a programming language is that it can increase the *understandability* of the programs written in the language. The reason is that modules can be separately understood, i.e., the programmer does not have to oversee the entire workings of the system when considering a certain module. Further, modularization enables *reuse* of software, since modules can potentially be used in different programs or parts of a single program. Finally, we mention an important principle which any kind of modularity should stick to, namely the principle of *information hiding*. This means that information about a module should be private to the module, i.e., not accessible from outside, unless it is specifically declared as public. The amount of information declared public should typically be relatively small. The idea behind information hiding is that a module can be changed (or at least the non-public part of a module), without affecting other modules.

In this chapter we address the issue of modularization in *cognitive agent programming languages*. Cognitive agents are agents endowed with high-level mental attitudes such as beliefs, goals, plans, etc. Several programming languages and platforms have been introduced to program these agents, such as 3APL [Hindriks et al., 1999b, Dastani et al., 2004], AgentSpeak(L) [Rao, 1996, Moreira and Bordini, 2002], JACK™ [Winikoff, 2005], Jadex [Pokahr et al., 2005b], etc. Some of these languages incorporate support for modularization, which we discuss in some detail in Section 9.1.1.

Our contribution is the proposal of a new kind of modularity, i.e., *goal-oriented modularity*, which is, as we will argue, particularly suited for agent programming languages (Section 9.1). Further, we present a *formalization* of goal-oriented modularity in the context of the 3APL programming language (Section 9.2). We conclude the chapter with directions for future research in Section 9.3.

9.1 Goal-Oriented Modularity

In this section, we explain the general idea of goal-oriented modularity. First, we discuss which kinds of modularity are present in today's cognitive agent programming languages and platforms (Section 9.1.1). Then, we explain the general idea of goal-oriented modularity (Section 9.1.2).

As for programming in general, modularization is also an important issue for agent programming. One could argue that the agent paradigm provides inherent support for modularity, since a complex problem can be broken down and solved by a team of autonomous agents. Constructing a team of agents to solve a problem rather than creating a single more complex agent, might however not always be the appropriate approach. The team approach will likely generate significant communication overhead, and having several independent agents can make it difficult to handle problems that require global reasoning [Busetta et al., 2000, Braubach et al., 2006]. Following [Busetta et al., 2000, Braubach et al., 2006], we thus argue that modularization is important also at the level of individual agents.

9.1.1 Related Work

With regard to cognitive agent programming languages such as 3APL and AgentSpeak(L), one could argue that these languages support modularization: an agent is typically composed of a number of components such as a belief base, a plan or intention base, a plan library, a goal or event base, etc. These components however do not provide the appropriate modularization, since their workings are closely intertwined and therefore they cannot be considered as relatively independent units of functionality. Cognitive agent programming languages thus need more support for structuring the internals of an agent, in order for them to deserve the predicate "modular".

One possible approach to addressing this issue of modularity of cognitive agent programming languages has been proposed in [Busetta et al., 2000]. In that paper, the notion of *capability* was introduced and its implementation in the JACK cognitive agent programming language was described.

JACK extends the Java™ [Gosling et al., 2000] programming language in several ways, such as by introducing constructs for declaring cognitive agent notions like beliefs, events, plans, etc. Events are used to model messages being

received, new goals being adopted, and information being received from the environment. Plans are used to handle events, i.e., if an event is posted, the reasoning engine tries to find an appropriate plan which has this event as its so-called triggering condition.

A capability in JACK is a cluster of components of a cognitive agent, i.e., it encapsulates beliefs, events (either posted or handled by the capability), and plans. Examples of capabilities given in [Busetta et al., 2000] are a buyer and seller capability, clustering functionality for buyer and seller agents, respectively. A capability can import another capability, in which case the latter becomes a sub-capability of the former. Using the importation mechanism for capabilities, the beliefs of a capability can also be used by its super-capability, that is, if they are explicitly imported by the latter. Also, the beliefs of a capability can be used by its sub-capabilities if they are explicitly declared as exported (and if they are also imported by the sub-capabilities). For events, a similar mechanism exists, by means of which events posted from one capability can also be handled by plans of its sub- and possibly super-capabilities.

The notion of capability as used in JACK has been extended in the context of the Jadex platform [Braubach et al., 2006]. Jadex is a cognitive reasoning engine which is built on top of the Jade [Bellifemine et al., 2000] agent platform. A Jadex agent has beliefs, goals, plans, and events. Like capabilities in JACK, a Jadex capability clusters a set of beliefs, goals, plans, and events. Its most important difference with the notion of capability as used in JACK, is the fact that a general import/export mechanism is introduced for all kinds of elements of a capability, i.e., the mechanism is the same for beliefs, events, etc.

Another approach which could be viewed as addressing the issue of modularity in cognitive agent programming languages, has been proposed in the context of 3APL in [Dastani et al., 2005b]. In that paper, a formalization of the notion of a role is given. Similar to capabilities, a role clusters beliefs, goals, plans, and reasoning rules. The usage of roles at run-time however differs from that of capabilities. In the cited paper, a role can be enacted and deacted at run-time, which is specified at the level of the 3APL reasoning engine or deliberation cycle. If a role is enacted, the agent pursues the goals of the role, using the plans and reasoning rules of the role.[1] Further, the agent has a single belief base, and if a role is enacted, the beliefs associated with the role are added to the agent's beliefs. This is in contrast with JACK and Jadex where beliefs are distributed over capabilities, and can only be used in other capabilities if explicitly imported and exported. Also, only one role at the time can be active. This is in contrast with the way capabilities are used, since a JACK or Jadex agent can in principle use any of its capabilities at all times.

[1] We simplify somewhat, since the details are not relevant for the purpose of this chapter.

9.1.2 Our Proposal

While the approaches to modularization as described in Section 9.1.1 are interesting in their own right, we propose an alternative which we argue to be particularly suited for cognitive agent programming languages. As the name suggests, goal-oriented modularity takes the goals of an agent as the basis for modularization. The idea is that *modules encapsulate the information on how to achieve a goal*, or a set of (related) goals. That is, modules contain information about the plans that can be used to achieve a (set of) goal(s). At run-time, the agent can then dispatch a goal to a module, which, broadly speaking, tries to achieve the dispatched goal using the information about plans contained in the module.

This mechanism of dispatching a goal to a module can be used for an agent's top-level goals[2], but also for the *subgoals* as occurring in the plans of an agent. Plans are often built from actions which can be executed directly, and subgoals which represent a state that is to be achieved before the agent can continue the execution of the rest of the plan. An agent can for example have the plan to take the bus into town, to achieve the goal of having bought a birthday cake, and then to eat the cake.[3] This goal of buying a birthday cake will have to be fulfilled by executing in turn an appropriate plan of for example which shops to go to, paying for the cake, etc., before the agent can execute the action of eating the cake.

Before continuing, we remark that goals and subgoals in this chapter are *declarative* goals, which means that they describe a state that is to be reached. This is in contrast with procedural goals, which are directly linked to courses of action. We refer to Chapter 5 (Section 5.4.1) for a discussion on declarative and procedural goals. In Chapter 3, semantics of subgoals are explored, and declarative and procedural interpretations of subgoals are related to one another.

Returning to our treatment of goal-oriented modularity, the idea is thus that agents try to achieve subgoals of a plan by dispatching the subgoal to an appropriate module, i.e., by *calling* a module. The module should then define the plans that can be used for achieving the (sub)goal. If a module is called for a goal, these plans are tried one by one until either the goal is achieved, or all plans have been tried. Control then returns to the plan from which the module was called. Depending on whether the subgoal is achieved or not upon returning from the module, the plan respectively continues execution, or fails. If the plan fails, another plan is selected (if it exists), for achieving the goal for which the failed plan was selected, etc.

[2]Top-level goals are goals that the agent, e.g., has from start-up, or that it for example has adopted because of requests from other agents, etc.

[3]Assuming that both taking the bus into town and eating cake are actions that can be executed directly.

9.1.3 Discussion

An advantage of our proposal is the *flexible* agent behavior with respect to handling plan failure, which comes with the usage of declarative goals. As argued in Section 5.4.1, the usage of declarative goals facilitates a decoupling of plan execution and goal achievement. If a plan fails, the goal that was to be achieved by the plan remains a goal of the agent, and the agent can select a different plan to try to achieve the goal. While these ideas regarding declarative goals are not new, we contribute by proposing to use modules to *encapsulate* this mechanism for achieving goals by trying different plans. We thus exploit the advantages of declarative goals for obtaining modularization. Since in our view goals, proactiveness and flexible behavior are at the heart of (cognitive) agenthood, we argue that goal-oriented modularity, which builds on these notions, is a kind of modularity *fitting* for cognitive agents.

Comparing goal-oriented modularity with capabilities, we point out two major differences. Firstly, in the case of capabilities, there is no notion of *calling* a capability for a subgoal, thereby passing control to another capability. An event (or subgoal) posted from the plan of one capability, will be handled by the plans of this capability itself. That is, unless the capability imports other capabilities, in which case the plans of these other capabilities are *added* to the set of plans considered for handling the posted event (given appropriate import and export declarations of events). This is thus in contrast with goal-oriented modularity, where, in case of calling a module, *only* the plans of the called module are considered. These plans are not added to, e.g., some other set of plans, thereby preventing possibly unforeseen interactions between these plans. One could thus consider the idea of goal-oriented modularity to provide a higher degree of modularity or encapsulation of behavior at run-time, compared with the way in which capabilities are used. As we will explain in Section 9.2, this is especially advantageous in the case of 3APL.

Secondly, modules in goal-oriented modularity contain only information about the plans which can be used to achieve certain goals. This is in contrast with capabilities, which can also contain beliefs. The idea for goal-oriented modularity is that the agent has one global belief base, rather than defining beliefs inside modules. When using capabilities, beliefs can by contrast be distributed over these capabilities, and only the beliefs of a certain capability can be accessed from this capability.

While from a software engineering perspective it might be convenient to be able to define beliefs inside capabilities (or modules), it can be considered less intuitive from a conceptual point of view. When testing, e.g., from within a plan whether the agent believes something, one could argue that the agent would have to consider *all* of its beliefs, rather than just the ones available in the capability. Also, if logical reasoning is involved as in the case of 3APL, it is more intuitive to let an agent have just one belief base. Consider for example that the formula p is in the belief base of one module, and that $p \rightarrow q$ is in the

belief base of another. When testing whether q holds from the latter module, one would probably want the test to succeed.

Nevertheless, in JACK and Jadex, beliefs can be used in other capabilities if they are imported and exported in appropriate ways. Also, beliefs (or beliefsets) in those languages are effectively databases which store elements representing the beliefs of the agent. The definition of beliefs inside a capability could thus be viewed as the specification of a *part* of the larger database (or set of databases) comprising the total set of beliefs of the agent. From within a capability, an agent can then only refer to the part of its beliefs defined in this capability. It is then up to the programmer to make sure that the beliefs of a capability are the only ones relevant for the plans of this capability. The possibility of storing beliefs inside modules in a way which is somewhere inbetween the current proposal and the way it is done for capabilities, is discussed in Section 9.3.

Comparing goal-oriented modularity with the notion of roles as used in [Dastani et al., 2005b], we remark the following. As in the case of capabilities, roles can, in contrast with modules, not call each other. Also, beliefs can be part of the definition of a role. However, a role does not have its own beliefs once it is enacted at run-time. If a role is enacted, its beliefs are added to the global belief base of the agent. This is more in line with goal-oriented modularity, but it is in contrast with the way capabilities are used. Further, contrary to roles, modules do not have goals. That is, a goal can be dispatched to a module, but goals are not part of the definition of a module.

9.2 Goal-Oriented Modularity in 3APL

In this section, we make the idea of goal-oriented modularity as presented in Section 9.1 precise. In particular, we present a formalization in the context of the language similar to the language of Chapter 2, to which we will simply refer as "3APL" in this chapter. Although this formalization is presented in the context of 3APL, we stress that we consider the general idea of goal-oriented modularity to be suited for cognitive agent programming languages in general, rather than for 3APL only.

9.2.1 Syntax

A 3APL agent has beliefs, a plan, goals, rules for selecting a plan to achieve a certain goal given a certain belief, and rules for revising its plan during execution. We use these ingredients for our formalization of goal-oriented modularity.

Throughout this chapter, we assume a language of propositional logic $\mathcal{L}$ with negation and conjunction. The symbol $\models$ will be used to denote the standard entailment relation for $\mathcal{L}$. Further, we assume a belief query language $\mathcal{L}_{\mathsf{B}}$ with typical element β (see Definitions 2.2 and 2.7 for the syntax and semantics of $\mathcal{L}_{\mathsf{B}}$).

Below, we define the language of plans Plan. A plan is a sequence of basic actions and module calls. Basic actions can change the beliefs of an agent if executed. A module call is of the form $m(\phi)$, where m is the name of a module (to be defined in Definition 9.4), and ϕ is a propositional formula representing the goal which is dispatched to module m.

Further, we define an auxiliary set of plans Plan' with the additional construct $m(\phi)\downarrow$. This construct is used in the semantic definitions for recording whether module m has already been called for goal ϕ (see Section 9.2.2 for further explanation). It should not be used by the agent programmer for programming plans, which is why we define two different plan languages.

Definition 9.1 *(plan)* Let $\mathsf{BasicAction}$ with typical element a be the set of basic actions, let $\mathsf{ModName}$ with typical element m be a set of module names, and let $\phi \in \mathcal{L}$. The set of plans Plan with typical element π is then defined as follows.

$$\pi ::= a \mid m(\phi) \mid \pi_1; \pi_2$$

We use ϵ to denote the empty plan and identify $\epsilon;\pi$ and $\pi;\epsilon$ with π. The set Plan' is defined as follows.

$$\pi ::= a \mid m(\phi) \mid m(\phi)\downarrow \mid \pi_1; \pi_2$$

3APL uses rules for selecting an appropriate plan for a certain goal. In this chapter, these rules are called *plan generation rules*. A plan generation rule is of the form $\phi \mid \beta \Rightarrow \pi$. This rule represents that it is appropriate to select plan π for goal ϕ, if the agent believes β.[4]

Definition 9.2 *(plan generation rule)* The set of plan generation rules $\mathcal{R}_{\mathsf{PG}}$ is defined as follows: $\mathcal{R}_{\mathsf{PG}} = \{\phi \mid \beta \Rightarrow \pi \ : \ \phi \in \mathcal{L},\ \beta \in \mathcal{L}_{\mathsf{B}},\ \pi \in \mathsf{Plan}\}$.[5]

In contrast with the plan selection rules of Chapter 2 (Definition 2.4), plan generation rules in this chapter use a propositional formula for referring to the goal for which the rule may be applied, rather than a goal formula. This has to do with the fact that modules are called with a propositional formula.

Plan revision rules are as in Chapter 2 (Definition 2.4).

Definition 9.3 *(plan revision rule)* The set of plan revision rules $\mathcal{R}_{\mathsf{PR}}$ is defined as follows: $\mathcal{R}_{\mathsf{PR}} = \{\pi_h \mid \beta \rightsquigarrow \pi_b \ : \ \beta \in \mathcal{L}_{\mathsf{B}},\ \pi_h, \pi_b \in \mathsf{Plan}, \pi_h \neq \epsilon\}$.

Plan generation rules capture the information about which plan can be selected for which goal, and plan revision rules can be used during the execution of a plan. These rules thus specify the information on how to achieve goals, and therefore

[4]Note that it is up to the programmer to specify appropriate plans for a certain goal. 3APL agents can thus be viewed as a kind of reactive planning agents.

[5]We use the notation $\{\ldots \ : \ \ldots\}$ instead of $\{\ldots \mid \ldots\}$ to define sets, to prevent confusing usage of the symbol | in this definition and Definition 9.3.

we propose to have these rules make up a module, as specified below. It is important to remark that non-modular versions of 3APL have one set of plan generation rules, and one set of plan revision rules, rather than an encapsulation of these into modules.

Definition 9.4 *(module)* A module is a tuple $\langle m, \mathsf{PG}, \mathsf{PR} \rangle$, consisting of a module name m, a finite set of plan generation rules $\mathsf{PG} \subseteq \mathcal{R}_{\mathsf{PG}}$, and a finite set of plan revision rules $\mathsf{PR} \subseteq \mathcal{R}_{\mathsf{PR}}$.

The mechanism of calling a module is formalized using the notion of a stack. This stack can be compared with the stack resulting from procedure calls in procedural programming, or method calls in object-oriented programming. During execution of the agent, a single stack is built (see Definition 9.7). Each element of the stack represents, broadly speaking, a module call.

To be more specific, each element of the stack is of the form $(\phi, \pi, \mathsf{PG}, \mathsf{PR})$, where ϕ is the goal for which the module was called, π is the plan currently being executed in order to achieve ϕ, and PG and PR correspond with the plan generation and plan revision rules of the module which was called for achieving ϕ. Rather than using the name of the module to refer to the relevant plan generation and plan revision rules, we copy these rules into the stack element. The reason is that we want to remove plan generation rules if they are tried once. This will be explained further in Section 9.2.2.

Definition 9.5 *(stack)* The set of stacks Stack with typical element S to denote arbitrary stacks, and s to denote single elements of a stack, is defined as follows, where $\phi \in \mathcal{L}$, $\pi \in \mathsf{Plan}'$, $\mathsf{PG} \subseteq \mathcal{R}_{\mathsf{PG}}$, and $\mathsf{PR} \subseteq \mathcal{R}_{\mathsf{PR}}$.

$$\begin{array}{lcl} \mathrm{s} & ::= & (\phi, \pi, \mathsf{PG}, \mathsf{PR}) \\ \mathrm{S} & ::= & \mathrm{s} \mid \mathrm{s}.\mathrm{S} \end{array}$$

E is used to denote the empty stack (or the empty stack element), and E.S is identified with S.

Note that the plan π of a stack element is from the extended set of plans Plan', since a stack is a run-time construct which is not specified by the programmer when programming an agent (see Definition 9.6). The plan π might thus, in contrast with the plans of plan generation and plan revision rules, contain a construct of the form $m(\phi)\downarrow$. The empty stack E is introduced for technical convenience when defining the semantics in Section 9.2.2. Stacks as used here differ from intention stacks of AgentSpeak(L), as the elements comprising the stacks are essentially different: stack elements in this chapter correspond with module calls, whereas in AgentSpeak(L) they represent (parts of) plans or intentions. Like non-modular 3APL, AgentSpeak(L) has one large set of plans (corresponding with the rules of 3APL).

An agent, as defined below, consists of a belief base, a goal base, a set of modules, and a function which specifies how beliefs are updated if actions are

executed. As in non-modular 3APL, the belief base σ is a consistent set of propositional formulas. The goal base γ is essentially also a set of propositional formulas, and forms the top-level goals of the agent. In contrast with non-modular 3APL however, each goal is associated with a module which should be called for achieving the goal, i.e., goals are of the form $m(\phi)$. The set of modules Mod form the modules which can be called to achieve (sub)goals.

Definition 9.6 *(agent)* Let $\Sigma = \{\sigma \mid \sigma \subseteq \mathcal{L},\ \sigma \not\models \bot\}$ be the set of belief bases. An agent is a tuple $\langle \sigma, \gamma, \mathsf{Mod}, \mathcal{T} \rangle$ where $\sigma \in \Sigma$ is the belief base, $\gamma \subseteq \{m(\phi) \mid m \in \mathsf{ModName},\ \phi \in \mathcal{L}\}$ is the initial goal base, and Mod is a set of modules such that each module in this set has a distinct name. $\mathcal{T}$ is a partial function of type $(\mathsf{BasicAction} \times \Sigma) \rightarrow \Sigma$ and specifies the belief update resulting from the execution of basic actions.

The notion of a configuration, as defined below, is used to represent the state of an agent at each point during computation. It consists of the elements which may change during execution of the agent, i.e., it consists of a belief base, a goal base, and a stack. Note that an agent initially has an empty stack.

Definition 9.7 *(configuration)* A configuration is a tuple $\langle \sigma, \gamma, \mathrm{S} \rangle$ where $\sigma \in \Sigma$, $\gamma \subseteq \{m(\phi) \mid m \in \mathsf{ModName},\ \phi \in \mathcal{L}\}$, and $\mathrm{S} \in \mathsf{Stack}$. If $\langle \sigma, \gamma, \mathsf{Mod}, \mathcal{T} \rangle$ is an agent, then $\langle \sigma, \gamma, E \rangle$ is the initial configuration of the agent.

9.2.2 Semantics

The semantics of modular 3APL agents is defined by means of a transition system [Plotkin, 1981]. In the transition rules below, we assume an agent with a set of modules Mod, and a belief update function $\mathcal{T}$.

The first transition rule specifies how a transition for a composed stack can be derived, given a transition for a single stack element. It specifies that only the top element of a stack can be transformed or executed.[6]

Definition 9.8 *(stack execution)* Let $s \neq E$.

$$\frac{\langle \sigma, \gamma, \mathrm{s} \rangle \rightarrow \langle \sigma', \gamma', \mathrm{S}' \rangle}{\langle \sigma, \gamma, \mathrm{s.S} \rangle \rightarrow \langle \sigma', \gamma', \mathrm{S}'.\mathrm{S} \rangle}$$

In the transition rule for stack execution, we specify that s cannot be the empty stack. The reason is related to the transition rule for stack initialization of Definition 9.9 below. In that rule, we want to specify that a stack initialization transition can only be derived if the current stack is empty. We however do not want to use that rule in combination with the rule for stack execution, since that

[6] For technical convenience, we overload the "." operator in Definition 9.8. We use it to "push" a stack onto a stack, rather than to push a single stack element onto a stack, as it was, strictly speaking, defined in Definition 9.5.

would result in the possibility of deriving an "initialization" transition, even if the current stack is not empty.

An initialization transition can thus only be derived if the current stack is empty. The idea of initialization is that a (top-level) goal $m(\phi)$ from the goal base is (randomly) selected, and then the module m is called with the goal ϕ. The resulting stack is then of the form $(\phi, \epsilon, \mathsf{PG}, \mathsf{PR})$, where PG and PR are the plan generation and plan revision rules of m. Note that the plan of the resulting stack element is empty. The idea is that the plan generation rules of the module should now be used to generate an appropriate plan.

Definition 9.9 *(initialization of stack)*

$$\frac{m(\phi) \in \gamma \qquad \langle m, \mathsf{PG}, \mathsf{PR}\rangle \in \mathsf{Mod}}{\langle \sigma, \gamma, E\rangle \rightarrow \langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle}$$

Note that a consequence of this way of defining the goal base is that multiple goals from the goal base cannot be dispatched to the same module in one initialization. Thinking about alternative ways to define the goal base which might allow this, is left for future research.

The following transition rule is the rule for plan generation, defining when a plan generation rule can be applied. A prerequisite for applying a plan generation rule is that the plan of the relevant stack element is empty. A rule $\phi' \mid \beta \Rightarrow \pi$ from PG can then be applied if β is true, ϕ' follows from ϕ (i.e., the goal for which the module was called), and ϕ' is not believed to be achieved. Note that since ϕ' follows from ϕ, it is the case that ϕ is also not reached, if ϕ' is not reached. With respect to the second condition for plan generation rule application, we remark that a plan generation rule with goal antecedent p (i.e., $\phi' = p$) can thus be applied if the agent has, e.g., a goal $p \wedge q$ (i.e., $\phi = p \wedge q$). It is convenient when programming, since plan generation rules can be specified for parts of a composed goal.

If a plan generation rule is applied, the plan π in its consequent becomes the plan of the resulting stack element. Further, the applied plan generation rule is removed from the set of plan generation rules of the stack element.

Definition 9.10 *(plan generation)*

$$\frac{\phi' \mid \beta \Rightarrow \pi \in \mathsf{PG} \qquad \sigma \models_{\mathcal{L}_\mathsf{B}} \beta \qquad \phi \models \phi' \qquad \sigma \not\models \phi'}{\langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \pi, \mathsf{PG}', \mathsf{PR})\rangle}$$

where $\mathsf{PG}' = \mathsf{PG} \setminus \{\phi' \mid \beta \Rightarrow \pi\}$

The reason for removing plan generation rules is that we do not want the agent to try the same plan generation rule twice, to achieve a certain goal. Otherwise, the agent could get stuck "inside" a module trying to achieve a subgoal, if all its plans keep failing to reach the goal. The idea is that modules contain various

plans for achieving a certain goal. If the agent cannot reach a certain subgoal of a plan with the designated module, the agent should thus at a certain point give up trying to reach the subgoal. It should just try another plan with possibly different subgoals. Wanting to remove plan generation rules of a stack element is the reason that we copy the rules into stack elements, rather than just referring to the rules of a module using the name of the module. If we would extend this semantics to a first order version, we would have to record which instances of a plan generation rule are tried, rather than just removing the rule. This mechanism is comparable to the mechanism for selecting plans for subgoals in JACK. In JACK it is also the case that the same plan is not tried for the same subgoal twice.

The following two transition rules are standard for 3APL, and define how a plan is executed. The only difference is that the plan is now inside a stack element. Note that actions, which are executed from within a module, operate on the global belief base of the agent.

Definition 9.11 *(action execution)*

$$\frac{\mathcal{T}(a,\sigma) = \sigma'}{\langle \sigma, \gamma, (\phi, a; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma', \gamma', (\phi, \pi, \mathsf{PG}, \mathsf{PR})\rangle}$$

where $\gamma' = \gamma \setminus \{m(\phi) \mid \sigma \models \phi\}$.

The transition below specifies the application of a plan revision rule.

Definition 9.12 *(plan revision)*

$$\frac{\pi_h \mid \beta \rightsquigarrow \pi_b \in \mathsf{PR} \qquad \sigma \models_{\mathcal{L}_\mathsf{B}} \beta}{\langle \sigma, \gamma, (\phi, \pi_h; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \pi_b; \pi, \mathsf{PG}, \mathsf{PR})\rangle}$$

We now revisit the point made in Section 9.1.3, that the encapsulation provided by modules at run-time is especially advantageous in the case of 3APL. The reason is that modules encapsulate not only plan generation rules, but also plan revision rules. These plan revision rules provide very flexible ways for revising a plan during execution, but a large number of rules can interact in possibly unforeseen ways. This has to do with the semantics of plans not being compositional in case of plan revision, as discussed in Chapters 6, 7, and 8. Being able to cluster plan revision rules into modules thus reduces the chances of unforeseen interactions with other rules: the number of plan revision rules in a module will be small compared with the global set of plan revision rules of a non-modular 3APL agent.

The next two transition rules specify the cases in which a stack element can be popped from the stack. Both transition rules specify that an element can be popped if its plan has finished execution, i.e., if the plan is empty. The idea is, that an agent should always finish the execution of an adopted plan,

even though, e.g., the goal for which it was selected might already be reached. This is standard for 3APL, and the reason is that the programmer might have specified some necessary "clean-up" actions. Consider as an example the case where an agent still has to pay after refueling, even though the goal of having gas is already reached.

The first transition rule for popping a stack element specifies that the element can be popped if the goal ϕ of the stack element is reached.

Definition 9.13 *(goal of stack element reached)*

$$\frac{\sigma \models \phi}{\langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR}) \rangle \rightarrow \langle \sigma, \gamma, E \rangle}$$

The second transition rule for popping a stack element specifies that the element can be popped if there are no more applicable plan generation rules (regardless of whether the goal is reached). Since we assume that plan revision rules do not have the empty plan as a head (Definition 9.3), these rules cannot be applied if the plan of the stack element is empty.

Definition 9.14 *(no applicable plan generation rules)*

$$\frac{\neg\exists(\phi' \mid \beta \Rightarrow \pi) \in \mathsf{PG} : \ (\sigma \models_{\mathcal{L}_\mathsf{B}} \beta \text{ and } \phi \models \phi' \text{ and } \sigma \not\models \phi')}{\langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR}) \rangle \rightarrow \langle \sigma, \gamma, E \rangle}$$

The next three definitions specify the semantics of the construct $m(\phi')$ for calling a module. If this construct is encountered in a plan, and it is not annotated with the symbol $\downarrow$, the module m should be called for the goal ϕ'. That is, only if ϕ' is not yet reached. If a module with the name m exists,[7] a new element with goal ϕ', an empty plan, and the rules of m, is pushed onto the stack. Further, the construct $m(\phi')$ is annotated with $\downarrow$ to indicate that a module has been called for this subgoal. This is important, since we do not want to call module m again upon returning from m.

Definition 9.15 *(calling a module)*

$$\frac{\langle m, \mathsf{PG}', \mathsf{PR}' \rangle \in \mathsf{Mod} \qquad \sigma \not\models \phi'}{\langle \sigma, \gamma, (\phi, m(\phi'); \pi, \mathsf{PG}, \mathsf{PR}) \rangle \rightarrow \langle \sigma, \gamma, (\phi', \epsilon, \mathsf{PG}', \mathsf{PR}').(\phi, m(\phi')\downarrow; \pi, \mathsf{PG}, \mathsf{PR}) \rangle}$$

The transition rules of the next definition specify that the agent can continue the execution of the rest of its plan, if the subgoal ϕ' occurring at the head of the plan is reached. The agent should continue if it has already called the module m for the subgoal, i.e., if the construct is of the form $m(\phi')\downarrow$, or if the module has not yet been called, i.e., if the construct is of the form $m(\phi')$. The latter case is however probably less likely to occur.

[7] And it should if the programmer has done a good job.

Definition 9.16 *(subgoal reached)*

$$\frac{\sigma \models \phi'}{\langle \sigma, \gamma, (\phi, m(\phi')\downarrow; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \pi, \mathsf{PG}, \mathsf{PR})\rangle}$$

$$\frac{\sigma \models \phi'}{\langle \sigma, \gamma, (\phi, m(\phi'); \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \pi, \mathsf{PG}, \mathsf{PR})\rangle}$$

The next transition rule specifies that if a module has been called for a subgoal, i.e., if a construct of the form $m(\phi')\downarrow$ is at the head of the plan of a stack element, and the subgoal ϕ' has not been reached, the plan fails. That is, the plan is replaced by an empty plan. If there are any applicable plan generation rules left, another plan can then be selected to try to achieve the goal ϕ of the stack element.

Definition 9.17 *(subgoal dispatched and not reached)*

$$\frac{\sigma \not\models \phi'}{\langle \sigma, \gamma, (\phi, m(\phi')\downarrow; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle}$$

In this version of modular 3APL, a plan can fail to reach a goal only in case the programmer has specified the "wrong" actions in the plans. Since all actions (as is usually the case for 3APL's formal semantics) operate on the belief base and there is no notion of an external environment, there is no notion of the environment preventing an action from being executed, thereby possibly causing a plan to fail. One could thus argue that it is not necessary to have a mechanism for selecting a different plan upon plan failure, since it is the job of the programmer to make sure that the plans do not fail and reach the goals. We however aim for the ideas presented in this paper to be used in domains involving actions being executed in the environment, and it is thus important to incorporate a mechanism for handling plan failure.

The final transition rule below is very similar to the previous, in the sense that it specifies another reason for plan failure. In particular, it specifies that a plan fails if the action at its head cannot be executed, i.e., if the function $\mathcal{T}$ is undefined for the action and the belief base of the stack element.

Definition 9.18 *(failure of action execution)* Let "no applicable plan revision rule" be defined as $\neg\exists(\pi_h \mid \beta \rightsquigarrow \pi_b) \in \mathsf{PR} : \sigma \models_{\mathcal{L}_\mathsf{B}} \beta$ and $a;\pi = \pi_h;\pi'$ for some π'.

$$\frac{\text{no applicable plan revision rule} \qquad \mathcal{T}(a,\sigma) \text{ is undefined}}{\langle \sigma, \gamma, (\phi, a; \pi, \mathsf{PG}, \mathsf{PR})\rangle \rightarrow \langle \sigma, \gamma, (\phi, \epsilon, \mathsf{PG}, \mathsf{PR})\rangle}$$

9.2.3 Example

For illustration, we present a simple example of a modular 3APL agent. The agent has to bring a rock from the location where the rocks are $loc(rock)$[8], to its

[8]Note that all formulas are propositional, although we use brackets for presentation purposes.

base location $loc(base)$. It has three modules, i.e., *collectRock* for the general goal of collecting the rock, and $goTo$ and $pickUp$ for going to a location and picking up a rock, respectively. Initially, the agent believes that it is at the base location, and that it does not have a rock, i.e., $\neg have(rock)$. Further, it has the goal $collectRock(have(rock) \wedge loc(base))$, i.e., it wants to achieve the goal $have(rock) \wedge loc(base)$ using module *collectRock*.

Below, we define the initial belief base σ and goal base γ of the rock collecting agent.

$$\begin{array}{lcl} \sigma & = & \{loc(base), \neg have(rock)\} \\ \gamma & = & \{collectRock(have(rock) \wedge loc(base))\} \end{array}$$

The plan generation rules PG_{cr} of the module *collectRock* are defined as below. The set of plan revision rules is empty.

$$\begin{array}{lcl} \mathsf{PG}_{cr} & = & \{have(rock) \wedge loc(base) \mid \top \Rightarrow \\ & & \quad goTo(loc(rock)); pickUp(have(rock)); goTo(loc(base))\} \end{array}$$

There is one plan generation rule which can be used for the goal $have(rock) \wedge loc(base)$, i.e., the initial goal of the agent. The plan of the rule consists entirely of calls to other modules. Note that the $goTo$ module is called with two different goals. The rule does not specify a condition on beliefs (other than $\top$). In particular, there is no need to specify that the agent should, e.g., believe that it is at the base location, rather than at the rock location. If the agent would already be at the rock location, the first module call of the plan, i.e., $goTo(loc(rock))$, would be skipped, since the subgoal $loc(rock)$ is already reached (see Definition 9.16).

The plan generation rules PG_{gt} and plan revision rules PR_{gt} of the module $goTo$ are defined as follows.

$$\begin{array}{lcll} \mathsf{PG}_{gt} & = & \{loc(rock) & \mid \mathbf{B}(loc(base)) \Rightarrow toRock, \\ & & loc(base) & \mid \mathbf{B}(loc(rock)) \Rightarrow toBase\} \\ \mathsf{PR}_{gt} & = & \{toRock & \mid \mathbf{B}(loc(rock)) \rightsquigarrow skip, \\ & & toRock & \mid \neg\mathbf{B}(loc(rock)) \rightsquigarrow east; toRock, \\ & & toBase & \mid \mathbf{B}(loc(base)) \rightsquigarrow skip, \\ & & toBase & \mid \neg\mathbf{B}(loc(base)) \rightsquigarrow west; toBase\} \end{array}$$

This module has two plan generation rules: one for selecting a plan to get from the base location to the rock location, and one for the other way around. The plans *toRock* and *toBase* in the bodies of the plan generation rules are non-executable basic actions, which are used as procedure variables.[9] Assuming that the rock is located east from the base location, we specify plan revision rules for moving east until the rock location is reached, and moving west until the base location is reached. The action *skip* is a special action which does nothing if executed.

[9]Non-executable basic actions are often termed *abstract plans* in the 3APL literature.

In this simple example, we specify separate plan revision rules for moving east and west respectively. The plans for moving to the rock location and to the base location thus use different plan revision rules. We could therefore have created two separate modules, i.e., one for going to the rock, and one for going to the base. In a more realistic setting however, one could imagine to have *one* set of plan revision rules for moving to any given location. In that case, it would be advantageous to specify these rules only in one module.

The plan generation rules PG_{pu} of the module *pickUp* are defined as below, and the set of plan revision rules is empty.

$$\begin{aligned}\mathsf{PG}_{pu} \quad = \quad & \{have(rock) \mid \mathbf{B}(loc(rock)) \Rightarrow pickUp1, \\ & \ have(rock) \mid \mathbf{B}(loc(rock)) \Rightarrow pickUp2\}\end{aligned}$$

The plan generation rules of this module can be applied if the agent believes it is at the rock location. Note that the call $pickUp(have(rock))$ to this module from the *collectRock* module, can only be executed if the subgoal of the previous module call, i.e., $goTo(loc(rock))$, has been achieved. If $pickUp(have(rock))$ is executed, we thus know for sure that the agent believes to be at the rock location. The module *pickUp* illustrates that the agent may have multiple plans, i.e., $pickUp1$ and $pickUp2$ in this case, for achieving the same goal. If, e.g., $pickUp1$ fails, the agent can try $pickUp2$.

9.3 Future Research

As directions of future research, we mention the following. In the current proposal there is no notion of importing a module into another module. Some notion of importation could be added in a straightforward way, but it will have to be investigated what exactly the semantics of importation should be. Should the rules of the imported module just be "added" to the other module, or should there be some kind of prioritization among rules of the module itself and the imported module?

Further, goal-oriented modularity as presented here provides a high degree of information hiding. That is, when calling a module, only the name of the module has to be known. In principle, any part of a module can be adapted without having to adapt the call to the module. We thus have no notion of an interface of a module, i.e., those parts of the module which are known outside the module. Nevertheless, it might be worthwile to investigate whether the framework can be extended with some notion of interface, such as the goals for which a module can be called, etc. This might be a useful tool to help a programmer.

Also, we envisage that a mechanism similar to the mechanism of dispatching a goal to a module, could be used in a multi-agent team work setting (see, e.g., [Yoshimura et al., 2000]) to delegate a goal to an agent. Using a uniform mechanism both for calling modules and delegating goals to agents could potentially

yield more transparent systems. In case of delegation, a plan would have to contain a request message to an agent, rather than a call to a module. A way of "returning" from the request, just as one can return from a module, would have to be defined. An agent would for example have to report back to the requesting agent, either with a message expressing that the goal has been achieved, or that he has failed and stopped trying. Moreover, it can be interesting to investigate a construct for calling modules in parallel. This can also be interesting in the case of multi-agent teamwork as also discussed in [Yoshimura et al., 2000], since an agent could then dispatch several goals to different agents in parallel.

Finally, we remark that it could be useful to be able to store information during execution within a module, which would not need to be kept after returning from the module. This could be realized by adding actions to the specification of a module, which would then update the module's internal information store, rather than the global belief base. This could be considered as a compromise between the way beliefs are handled in capabilities, and the way we handle beliefs in the presented framework.

Concluding, we have presented the idea of goal-oriented modularity, which takes the goals of an agent as the basis for modularization. Since we view goals as being an essential ingredient of cognitive agents, we argue that this approach to modularity is suited for cognitive agent programming languages. Further, we have shown how goal-oriented modularity might be incorporated in a cognitive agent programming language, by presenting a formalization of goal-oriented modularity in 3APL.

Acknowledgements

We would like to thank Alexander Pokahr, Lin Padgham, and Lars Braubach for very useful discussions on the issue of modularization in cognitive agent programming frameworks.

Chapter 10

Prototyping 3APL in the Maude Term Rewriting Language

This chapter is based on [van Riemsdijk et al., 2006b]. Research on agent programming languages is concerned with an investigation of what kind of programming constructs an agent programming language should contain, and what exactly the meaning of these constructs should be. In order to test whether these constructs indeed facilitate the programming of agents in an effective way, the programming language has to be implemented.

This can be done using Java, which was for example used for implementing the agent programming language 3APL [Hindriks et al., 1999b] [Dastani et al., 2004]. Java has several advantages, such as its platform independence, its support for building graphical user interfaces, and the extensive standard Java libraries. A disadvantage is however that the translation of the formal semantics of an agent programming language such as 3APL into Java is not very direct. It can therefore be difficult to ascertain that such an implementation is a faithful implementation of the semantics of the agent programming language, and experimenting with different language constructs and semantics can be quite cumbersome.

As an alternative to the use of Java, we explore in this chapter the usage of the Maude term rewriting language [Clavel et al., 2005] for prototyping 3APL. Maude is based on the mathematical theory of rewriting logic. The language has been shown to be suitable both as a *logical* framework in which many other logics can be represented, and as a *semantic* framework, through which programming languages with an operational semantics can be implemented in a rigorous way [Martí-Oliet and Meseguer, 2000]. We argue that, since agent programming languages such as 3APL have both a logical and a semantic component, Maude is very well suited for prototyping such languages (see Section 10.4.1). Further, we show that, since Maude is reflective, 3APL's meta-level rea-

soning cycle or deliberation cycle can be implemented very naturally in Maude (Section 10.3.2).

An important advantage of Maude is that it can be used for verification as it comes with an LTL model checker [Eker et al., 2002]. This chapter does not focus on model checking 3APL using Maude. However, the usage of Maude's model checker is relatively easy, given the implementation of 3APL in Maude. Nevertheless, the fact that Maude provides these verification facilities is an important, and was in fact, our original, motivation for our effort of implementing 3APL in Maude.

The outline of this chapter is as follows. We present (a simplified version of) 3APL in Section 10.1, and we briefly explain Maude in Section 10.2. We explain how we have implemented this simplified version of 3APL in Maude in Section 10.3. In Section 10.4, we discuss in more detail the advantages of Maude for the implementation of agent programming languages such as 3APL, and we address related work.

10.1 3APL

The cognitive agent programming language we have implemented in Maude comes close to the language of Chapter 2, and can be viewed as a propositional and otherwise slightly simplified version of the language 3APL as presented in [Dastani et al., 2004]. We will in the rest of this chapter refer to the language under consideration simply as "3APL".

We have implemented this simple version of 3APL to serve as a proof-of-concept of the usage of Maude for prototyping languages such as 3APL. In Section 10.4.2, we discuss the possible implementation of various extensions of the version of 3APL as defined in this section, although implementing these is left for future research.

10.1.1 Syntax

The version of 3APL as presented in this chapter takes a simple language, consisting of a set of propositional atoms, as the basis for representing beliefs and goals.

Definition 10.1 *(base language)* The base language is a set of atoms Atom.

As will be specified in Definition 10.7, the belief base and goal base are sets of atoms from Atom. This is thus a simplification of the representation of goals and beliefs of Chapter 2 (Definition 2.1), since we do not use arbitrary propositional formulas for their representation. The possibility of extending the representation of beliefs and goals and the accompanying belief and goal formulas is discussed in Section 10.4.2.

The language of plans is as in Chapter 2 (Definition 2.3), except that we omit abstract plans. Extending the implementation to include abstract plans is straightforward.

Definition 10.2 *(plan)* Let BasicAction with typical element a be the set of basic actions. The set of plans Plan with typical element π is then defined as follows.

$$\pi ::= a \mid \pi_1; \pi_2$$

We use ϵ to denote the empty plan and identify $\epsilon; \pi$ and $\pi; \epsilon$ with π.

The belief and goal formulas are as in Chapter 2 (Definition 2.2), except that the **B** and **G** operators take atoms as their arguments, rather than arbitrary propositional formulas. This is done for reasons of simplicity. In this chapter, we refer to these languages as *belief and goal query languages*. We explicitly include disjunction in the languages, for ease of representation.

Definition 10.3 *(belief and goal query language)* Let $p \in \mathsf{Atom}$. The belief query language $\mathcal{L}_\mathsf{B}$ with typical element β, and the goal query language $\mathcal{L}_\mathsf{G}$ with typical element κ, are then defined as follows.

$$\beta ::= \top \mid \mathbf{B}(p) \mid \neg\beta \mid \beta_1 \wedge \beta_2 \mid \beta_1 \vee \beta_2$$
$$\kappa ::= \top \mid \mathbf{G}(p) \mid \neg\kappa \mid \kappa_1 \wedge \kappa_2 \mid \kappa_1 \vee \kappa_2$$

The actions of an agent's plan update the agent's beliefs, if executed. In order to specify how actions should update the beliefs, we use so-called action specifications. Throughout this thesis and generally in the context of 3APL, often a belief update function $\mathcal{T}$ is assumed for this purpose, i.e., the exact definition of $\mathcal{T}$ is usually omitted. Since in this chapter we are concerned with implementing 3APL, we also have to be specific about the implementation of belief update through actions.

An action specification is of the form $\{\beta\}a\{Add, Del\}$. Here, a represents the action name, β is a belief query that represents the precondition of the action, and *Add* and *Del* are sets of atoms, that should be added to and removed from the belief base, respectively, if a is executed. This way of specifying how actions update beliefs corresponds closely with the way it is implemented in the Java version of 3APL.

Definition 10.4 *(action specification)* The set of action specifications $\mathcal{AS}$ is defined as follows: $\mathcal{AS} = \{\{\beta\}a\{Add, Del\} \ : \ \beta \in \mathcal{L}_\mathsf{B}, a \in \mathsf{BasicAction}, Add \subseteq \mathsf{Atom}, Del \subseteq \mathsf{Atom}\}$.[1]

Plan selection rules and plan revision rules are as in Chapter 2 (Definition 2.4), although the syntax of plan selection rules differs slightly.

[1] We use the notation $\{\ldots : \ldots\}$ instead of $\{\ldots \mid \ldots\}$ to define sets, to prevent confusing usage of the symbol | in Definition 10.6.

Definition 10.5 *(plan selection rule)* The set of plan selection rules $\mathcal{R}_{\mathsf{PS}}$ is defined as follows: $\mathcal{R}_{\mathsf{PS}} = \{\beta, \kappa \Rightarrow \pi \ : \ \beta \in \mathcal{L}_{\mathsf{B}}, \kappa \in \mathcal{L}_{\mathsf{G}}, \pi \in \mathsf{Plan}\}$.

Definition 10.6 *(plan revision rule)* The set of plan revision rules $\mathcal{R}_{\mathsf{PR}}$ is defined as follows: $\mathcal{R}_{\mathsf{PR}} = \{\pi_h \mid \beta \rightsquigarrow \pi_b \ : \ \beta \in \mathcal{L}_{\mathsf{B}}, \pi_h, \pi_b \in \mathsf{Plan}\}$.

10.1.2 Semantics

The semantics of 3APL agents is defined by means of a transition system [Plotkin, 1981]. Configurations in this chapter consist of a belief base σ and a goal base γ which are both sets of atoms, a plan, and sets of plan selection rules, plan revision rules, and action specifications. Reasoning rules and action specifications need to be included in configurations for a proper implementation of 3APL in Maude. Nevertheless, we omit these from the configurations in the transition rules of this section, for reasons of presentation.

Definition 10.7 *(configuration)* A 3APL configuration is a tuple $\langle \sigma, \pi, \gamma, \mathsf{PS}, \mathsf{PR}, \mathsf{AS} \rangle$[2] where $\sigma \subseteq \mathsf{Atom}$ is the belief base, $\pi \in \mathsf{Plan}$ is the plan, $\gamma \subseteq \mathsf{Atom}$ is the goal base, $\mathsf{PS} \subseteq \mathcal{R}_{\mathsf{PS}}$ is a set of plan selection rules, $\mathsf{PR} \subseteq \mathcal{R}_{\mathsf{PR}}$ is a set of plan revision rules, and $\mathsf{AS} \subseteq \mathcal{AS}$ is a set of action specifications.

Programming a 3APL agent comes down to specifying its initial configuration.

Before moving on to defining the transition rules for 3APL, we define the semantics of belief and goal queries. The semantics of belief and goal queries can be defined in a simple way, due to the simplicity of the query languages. That is, a formula $\mathbf{B}(p)$ is true in a configuration iff p is in the belief base, and $\mathbf{G}(p)$ is true iff p is in the goal base. The semantics of negation, disjunction, and conjunction are defined in the standard way (see Chapter 2 (Definition 2.7)), and we omit this here.

Definition 10.8 *(belief and goal queries)*

$$\begin{array}{lcl} \langle \sigma, \pi, \gamma \rangle \models_{\mathcal{L}_{\mathsf{B}}} \mathbf{B}(p) & \Leftrightarrow & p \in \sigma \\ \langle \sigma, \pi, \gamma \rangle \models_{\mathcal{L}_{\mathsf{G}}} \mathbf{G}(p) & \Leftrightarrow & p \in \gamma \end{array}$$

In Chapter 2 (Definition 2.7), we have defined the semantics of atomic goal formulas $\mathbf{G}\phi$ such, that it is true iff ϕ follows from the goal base *and* ϕ is not believed. The idea is, that an agent should not have something as a goal which it already believes to be the case. Given the simple structure of the belief base and goal base that we use in this chapter, it is however possible to make sure that an atom is never in the goal base if it is also in the belief base. We thus do not have to add this extra condition to the semantics of goal queries in this case.

[2]Note that in this chapter, the second element of a configuration is the plan, and the third element is the goal base. This differs from Chapter 2 (Definition 2.6).

The first transition rule as specified below is used to derive a transition for action execution. An action a that is the first action of the plan, can be executed if there is an action specification for a, and the precondition of this action as specified in the action specification holds. The belief base σ is updated such that the atoms of *Add* are added to, and the atoms of *Del* are removed from σ. Further, the atoms that have been added to the belief base should be removed from the goal base, as the agent believes these goals to be achieved. Also, the action is removed from the plan.

Definition 10.9 *(action execution)*

$$\frac{\{\beta\}a\{Add, Del\} \in \mathsf{AS} \qquad \langle \sigma, a; \pi, \gamma\rangle \models_{\mathcal{L}_\mathsf{B}} \beta}{\langle \sigma, a; \pi, \gamma\rangle \rightarrow \langle \sigma', \pi, \gamma'\rangle}$$

where $\sigma' = (\sigma \cup Add) \setminus Del$, and $\gamma' = \gamma \setminus Add$.

The semantics of the application of reasoning rules is as in Chapter 2 (Definitions 2.10 and 2.11).

Definition 10.10 *(plan selection rule application)*

$$\frac{\beta, \kappa \Rightarrow \pi \in \mathsf{PS} \qquad \langle \sigma, \epsilon, \gamma\rangle \models_{\mathcal{L}_\mathsf{B}} \beta \qquad \langle \sigma, \epsilon, \gamma\rangle \models_{\mathcal{L}_\mathsf{G}} \kappa}{\langle \sigma, \epsilon, \gamma\rangle \rightarrow \langle \sigma, \pi, \gamma\rangle}$$

Definition 10.11 *(plan revision rule application)*

$$\frac{\pi_h \mid \beta \rightsquigarrow \pi_b \in \mathsf{PR} \qquad \langle \sigma, \pi_h; \pi, \gamma\rangle \models_{\mathcal{L}_\mathsf{B}} \beta}{\langle \sigma, \pi_h; \pi, \gamma\rangle \rightarrow \langle \sigma, \pi_b; \pi, \gamma\rangle}$$

10.2 Maude

We cite from [Ölveczky, 2005]: "Maude is a formal declarative programming language based on the mathematical theory of rewriting logic [Meseguer, 1992]. Maude and rewriting logic were both developed by José Meseguer. Maude is a state-of-the-art formal method in the fields of algebraic specification [Wirsing, 1990] and modeling of concurrent systems. The Maude language specifies rewriting logic theories. Data types are defined algebraically by equations and the dynamic behavior of a system is defined by rewrite rules which describe how a part of the state can change in one step."

A rewriting logic specification consists of a *signature*, a set of *equations*, and a set of *rewrite rules*. The signature specifies the *terms* that can be rewritten using the equations and the rules. Maude supports membership equational logic [Meseguer, 1997], which is an extension of order-sorted equational logic, which is in turn an extension of many-sorted equational logic. For this chapter, it suffices to treat only the many-sorted subset of Maude. A signature in many-sorted equational logic consists of a set of *sorts*, used to distinguish different types of values, and a set of *function symbols* declared on these sorts.

In Maude, sorts are declared using the keyword `sort`, for example as follows: `sort List`. Function symbols can be declared as below, using the keywords `op` and `ops`.

```
op app :  Nat List -> List .
ops 0 1 2 3 :  -> Nat .
op nil :  -> List .
```

The function `app`, expressing that natural numbers can be appended to form a list, takes an argument of sort `Nat` and an argument of sort `List`, and the resulting term is again of sort `List`. The functions `0`, `1`, `2` and `3` are nullary functions, i.e., constants, of sort `Nat`. The nullary function `nil` represents the empty list. An example of a term (of sort `List`) over this signature is `app(1,app(2,app(3,nil)))`.

In order to define functions declared in the signature, one can use equations. An equation in Maude has the general form `eq` ⟨`Term-1`⟩ `=` ⟨`Term-2`⟩. Assume a function declaration `op sum : List -> Nat`, and a function `+` for adding natural numbers (declared as `op _+_ : Nat Nat -> Nat`, where the underscores are used express infix use of `+`). Further, assume variable declarations `var N : Nat` and `var L : List`, expressing that `N` and `L` are variables of sorts `Nat` and `List` respectively. The equations `eq sum(app(N,L)) = N + sum(L)` and `eq sum(nil) = 0` can then be used to define the function `sum`.

Maude also supports conditional equations, which have the following general form.

```
ceq ⟨Term-1⟩ = ⟨Term-2⟩
   if ⟨EqCond-1⟩ /\ ... /\ ⟨EqCond-n⟩
```

A condition can be either an ordinary equation of the form `t = t'`, a matching equation of the form `t := t'`, or an abbreviated boolean equation of the form `t`, which abbreviates `t = true`. An example of the use of a matching equation as the condition of a conditional equation is `ceq head(L) = N if app(N,L') := L`. This equation defines the function `head`, which is used to extract the first element of a list of natural numbers. The matching equation `app(N,L') := L` expresses that `L`, as used in the left-hand side of the equation, has to be of the form `app(N,L')`, thereby binding the first element of `L` to `N`, which is then used in the righthand side of the equation.

Operationally, equations can be applied to a term from left to right. Equations in Maude are assumed to be terminating and confluent, i.e., there is no infinite derivation from a term `t` using the equations, and if `t` can be reduced to different terms `t1` and `t2`, there is always a term `u` to which both `t1` and `t2` can be reduced. This means that any term has a *unique normal form*, to which it can be reduced using equations in a finite number of steps.

Finally, we introduce rewrite rules. A rewrite rule in Maude has the general form `rl [`⟨`Label`⟩`] :` ⟨`Term-1`⟩ `=>` ⟨`Term-2`⟩, expressing that term `Term-1`

can be rewritten into term `Term-2`. Conditional rewrite rules have the following general form.

```
crl [⟨Label⟩] ⟨Term-1⟩ => ⟨Term-2⟩
   if ⟨Cond-1⟩ /\ ... /\ ⟨Cond-n⟩
```

Conditions can be of the type as used in conditional equations, or of the form `t => t'`, which expresses that it is possible to rewrite term `t` to term `t'`. An example of a rewrite rule is `rl [duplicate] : app(N,L) => app(N,app(N,L))`, which expresses that a list with head `N` can be rewritten into a new list with `N` duplicated. The term `app(1,app(2,app(3,nil)))` can for example be rewritten to the term `app(1,app(1,app(2,app(3,nil))))` using this rule. The former term can however also be rewritten into `app(1,app(2,app(2,app(3,nil))))`, because rewrite rules (and equations alike) can be applied to subterms.

The way the Maude interpreter executes rewriting logic specifications, is as follows [Ölveczky, 2005]. Given a term, Maude tries to apply equations from left to right to this term, until no equation can be applied, thereby computing the normal form of a term. Then, an applicable rewrite rule is arbitrarily chosen and applied (also from left to right). This process continues, until no rules can be applied. Equations are thus applied to reduce each intermediate term to its normal form before a rewrite rule is applied.

Finally, we remark that in Maude, rewriting logic specifications are grouped into modules with the following syntax: `mod` ⟨`Module-Name`⟩ `is` ⟨`Body`⟩ `endm`. Here, ⟨`Body`⟩ contains the sort and variable declarations and the (conditional) equations and rewrite rules.

10.3 Implementation of 3APL in Maude

In this section, we describe how we have implemented 3APL in Maude. We distinguish the implementation of 3APL as defined in Section 10.1, which we will refer to as object-level 3APL (Section 10.3.1), and the implementation of a meta-level reasoning cycle (Section 10.3.2).

10.3.1 Object-Level

The general idea of the implementation of 3APL in Maude, is that 3APL configurations are represented as terms in Maude, and the transition rules of 3APL are mapped onto rewrite rules of Maude. This idea is taken from [Verdejo and Martí-Oliet, 2003], in which, among others, implementations in Maude of the operational semantics of a simple functional language and an imperative language are discussed. In this section we describe in some detail how we have implemented 3APL, thereby highlighting 3APL-specific issues.

Syntax

Each component of 3APL's syntax as specified in Definitions 10.1 through 10.6 is mapped onto a module of Maude. As an example, we present the definition of the module for the belief query language, corresponding with Definition 10.3.

```
mod BELIEF-QUERY-LANGUAGE is
    including BASE-LANGUAGE .

    sort BQuery .

    op B : LAtom -> BQuery .
    op top :  -> BQuery .
    op  ~_  :  BQuery -> BQuery .
    op  _/\_  :  BQuery BQuery -> BQuery .
    op  _\/_  :  BQuery BQuery -> BQuery .

endm
```

The module `BELIEF-QUERY-LANGUAGE` imports the module used to define the base language of Definition 10.1. A sort `BQuery` is declared, representing elements from the belief query language. The sort `LAtom` is declared in the module `BASE-LANGUAGE`, and represents atoms from the base language. Five operators are defined for building belief query formulas, which correspond with the operators of Definition 10.3. The other syntax modules are defined in a similar way. Note that only sort and function declarations are used in syntax modules. None of the syntax modules contain equations or rewrite rules.

The notion of configuration as specified in Definition 10.7 is also mapped onto a Maude module. This module imports the other syntax modules, and declares a sort `Conf` and an operator `op <_,_,_,_,_,_> :  BeliefBase Plan GoalBase PSbase PRbase ASpecs -> Conf`.

Semantics

The implementation of the semantics of 3APL in Maude can be divided into the implementation of the *logical* part, i.e., the belief and goal queries as specified in Definition 10.8, and the *operational* part, i.e., the transition rules of Definitions 10.9 through 10.11. The logical part, i.e., the semantics of the satisfaction relations $\models_{\mathcal{L}_B}$ and $\models_{\mathcal{L}_G}$, is modeled as equational specifications, whereas the transition rules of the operational part are translated into rewrite rules.

As an example of the modeling of the logical part, we present part of the module for the semantics of $\models_{\mathcal{L}_B}$ below. Here `[owise]` is a built-in Maude construct that stands for "otherwise".

```
mod BELIEF-QUERY-SEMANTICS is
    including BELIEF-QUERY-LANGUAGE .

    op  _|=LB_  :  BeliefBase BQuery -> Bool .

    var p :  LAtom .
    vars BB BB' : BeliefBase .
    vars BQ : BQuery .

    ceq BB |=LB B(p) = true if p BB' := BB .
    eq BB |=LB B(p) = false [owise] .

    ceq BB |=LB ~BQ = true if not BB |=LB BQ .
    eq BB |=LB ~BQ = false [owise] .
    ...

endm
```

The relation $\models_{\mathcal{L}_B}$ is modelled as a function `|=LB`, which takes a belief base of sort `BeliefBase` (a sort from the base language module), and a belief query of sort `BQuery`, and yields a boolean, i.e., `true` or `false`. Although the semantics of belief queries as specified in Definition 10.8 is defined on configurations rather than on belief bases, it is in fact only the belief base part of the configuration that is used in the semantic definition. For ease of specification we thus define the function `|=LB` on belief bases, rather than on configurations.

The first pair of (conditional) equations defines the semantics of a belief query `B(p)`. The matching equation `p BB' := BB` expresses that beliefbase `BB` is of the form `p BB'` for some beliefbase `BB'`,[3] i.e., that the atom `p` is part of `BB`. The second pair of (conditional) equations specifies the semantics of a negative query `~BQ`. The term `not BB |=LB BQ` is an abbreviated boolean equation, i.e., it abbreviates `not BB |=LB BQ = true`, and `not` is a built-in boolean connective. The module for the semantics of goal query formulas is defined in a similar way.

We now move on to the implementation of the operational part. Below, we present the rewrite rule for action execution, corresponding with the transition rule of Definition 10.9. The variables `B`, `B'` and `B''` are of sort `BeliefBase`, `A` is of sort `Action`, `P` is of sort `Plan`, and `G` and `G'` are of sort `GoalBase`. Moreover, `PSB`, `PRB`, and `AS` are respectively of sorts `PSbase`, `PRbase`, and `ASpecs`. Further, `Pre` is of sort `BQuery` and `Add` and `Del` are of sort `AtomList`.

```
crl [exec] : < B, A ; P, G, PSB, PRB, AS > =>
             < B', P, G', PSB, PRB, AS >
   if {Pre} A {Add,Del} AS' := AS  /\  B |=LB Pre /\ B'' := B U Add  /\
      B' := B'' \ Del  /\  G' := G \ Add .
```

[3] Belief bases are defined as associative and commutative space-separated sequences of atoms.

The transition as specified in the conclusion of the transition rule of Definition 10.9 is mapped directly to the rewrite part of the conditional rewrite rule.[4] The conditions of the transition rule, and the specification of how belief base and goal base should be changed, are mapped onto the conditions of the rewrite rule.

The first condition of the rewrite rule corresponds with the first condition of the transition rule. It specifies that if action `A` is to be executed, there should be an action specification for `A` in the set of action specifications `AS`. The second condition of the rewrite rule corresponds with the second condition of the transition rule, and specifies that the precondition of the action should hold. Note that the previously defined satisfaction relation `|=LB` is used here.

The third and fourth conditions of the rewrite rule specify how the belief base is changed, if the action `A` is executed. For this, a function `U` (union) has been defined using equations, which we omit here. This function takes a belief base and a list of atoms, and adds the atoms of the list to the belief base, thereby making sure that no duplicate atoms are introduced in the belief base. The function `\` for deleting atoms is defined in a similar way, and is also used for updating the goal base as specified in the last condition.

The translation of the transition rules for plan selection and plan revision rule application is done in a similar way. As an illustration, we present the rewrite rule for plan revision, corresponding with the transition rule of Definition 10.11. The variables `Ph` and `Pb` are of sort `Plan`, and `PRB'` is of sort `PRbase`. The syntax `(Ph | BQ -> Pb)` is used for representing a plan revision rule of the form $\pi_h \mid \beta \rightsquigarrow \pi_b$.

```
crl [apply-pr] :  < B, Ph ; P, G, PSB, PRB, AS > =>
                    < B, Pb ; P, G, PSB, PRB, AS >
   if (Ph | BQ -> Pb) PRB' := PRB  /\  B |=LB BQ .
```

As was the case for action execution, the transition as specified in the conclusion of the transition rule of Definition 10.11 is mapped directly onto the rewrite part of the conditional rewrite rule. The conditions of the transition rule furthermore correspond to the conditions of the rewrite rule.

Above, we have discussed the Maude modules for specifying the syntax and semantics of 3APL. In order to run a concrete 3APL program using Maude, one has to create another module for this program. In this module, one needs to specify the initial belief base, goal base, etc. For this, the atoms as can be used in, e.g., the belief base have to be declared as (nullary) operators of sort `LAtom`. Also, the possible basic actions have to be declared. Then, the initial configuration has to be specified. This can be conveniently done by declaring an operator for each component of the configuration, and specifying the value of that component using an equation. An initial belief base containing the atoms p and q can for example be specified using `eq bb = p q`, where `bb` is a nullary

[4] Recall that the plan selection and plan revision rule bases and the action specifications were omitted from Definitions 10.9 through 10.11 for reasons of presentation.

operator of sort `BeliefBase`, and `p` and `q` are atoms. In a similar way, the initial plan, goal base, rule bases, and action specifications can be defined. The 3APL program as thus specified can be executed by calling Maude with the command `rewrite <bb, plan, gb, psb, prb, as>`, where `<bb, plan, gb, psb, prb, as>` is the initial configuration.

10.3.2 Meta-Level

Given the transition system of 3APL as defined in Section 10.1.2, different possible executions might be derivable, given a certain initial configuration. It might for example be possible to execute an action in a certain configuration, as well as to apply a plan revision rule. The transition system does not specify which transition to choose during the execution. An implementation of 3APL corresponding with this transition system might non-deterministically choose a possible transition. The implementation of 3APL in Maude does just this, as Maude arbitrarily chooses an applicable rewrite rule for application.

In some cases however, it can be desirable to have more control over the execution. This can be achieved by making it possible to specify more precisely which transition should be chosen, if multiple transitions are possible. In the case of 3APL, meta-level languages have been introduced for this purpose (see [Hindriks et al., 1999b, Dastani et al., 2003] and Chapter 6). These meta-languages have constructs for specifying that an action should be executed or that a rule should be applied. Using a meta-language, various so-called *deliberation cycles* can be programmed.

A deliberation cycle can for example specify that the following process should be repeated: first apply a plan selection rule (if possible), then apply a plan revision rule, and then execute an action. Alternatively, a deliberation cycle could for example specify that a plan revision rule can only be applied if it is not possible to execute an action. It might depend on the application which is an appropriate deliberation cycle.

It turns out that this kind of meta-programming can be modeled very naturally in Maude, since rewriting logic is *reflective* [Clavel and Meseguer, 1996]. "Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object level representation correctly simulates the relevant metatheoretic aspects." [Clavel et al., 2005, Chapter 10]

In order to perform meta-level computation, terms and modules of the object-level have to be represented as Maude terms on the meta-level, i.e., they have to be *meta-represented*. For this, Maude has predefined modules, that include the functions `upTerm` and `upModule` for meta-representing terms and modules, and the function `metaXapply` which defines the meta-level application of a rewrite rule[5].

[5]Maude also has a function `metaApply` for this purpose with a slightly different meaning [Clavel et al., 2005]. It is however beyond the scope of this chapter to explain the difference.

The function `upTerm` takes an (object-level) term and yields the meta-representation of this term, i.e., a term of sort `Term`. The function `upModule` takes the meta-representation of the name of a module, i.e., the name of a module with a quote prefixed to it, and yields the meta-representation of the module with this name, i.e., a term of sort `Module`. The function `metaXapply` takes the meta-representation of a module, the meta-representation of a term, the meta-representation of a rule label (i.e., the rule label with a quote prefixed to it), and some more arguments which we do not go into here, as they are not relevant for understanding the general idea. The function tries to rewrite the term represented by its second argument using the rule as represented by its third argument. A rule with the label as given as the third argument of the function, should be part of the module as represented by the function's first argument. The function returns a term of sort `Result4Tuple?`. If the rule application was successful, i.e., if the rule could be applied to the term, the function returns a 4-tuple of sort `Result4Tuple`,[6] which contains, among other information, the term resulting from the rule application. This term can be retrieved from the tuple using the function `getTerm`, which returns the meta-representation of the term of sort `Term` resulting from the rewrite rule application.

Meta-level function calls, such as the application of a certain rewrite rule through `metaXapply`, can be combined to form so-called *strategies* [Clavel et al., 2005]. These strategies can be used to define the execution of a system at the meta-level. Deliberation cycles of 3APL can be programmed as these strategies.

An example of a deliberation cycle implemented in Maude is the following, which first tries to apply a plan selection rule, then to execute an action and then to apply a plan revision rule. The code below only specifies one sequence of rule applications and action executions. This sequence can be repeated to form a deliberation cycle using another function and equation, but we omit that code.

```
ceq one-cycle(Meta-Conf, Meta-Prog) = Meta-Conf'
   if Meta-Conf' := try-meta-apply-pr(try-meta-exec(try-meta-apply-ps(
                    Meta-Conf, Meta-Prog), Meta-Prog), Meta-Prog) .
```

Here, `Meta-Conf` and `Meta-Conf'` are variables of sort `Term` which stand for the meta-representations of 3APL configurations, and `Meta-Prog` is a variable of sort `Module`, which should be instantiated with the meta-representation of the module with Maude code of a 3APL program. The variable `Meta-Conf` is input to the function `one-cycle`, and `Meta-Conf'` represents the result of applying the function `one-cycle` to `Meta-Conf` and `Meta-Prog`. This module imports the syntax and semantics modules, which are also meta-represented in this way. The functions `try-meta-apply-pr`, `try-meta-exec`, and `try-meta-apply-ps`

[6] Note that the difference with the sort `Result4Tuple?` is the question mark. The sort `Result4Tuple` is a subsort of the sort `Result4Tuple?`.

try to apply the (object-level) rewrite rules for plan revision, action execution, and plan selection, respectively. In the definitions of these functions, the predefined function `metaXapply` is called, with the names of the respective object-level rewrite rules as one of its arguments, i.e., with, respectively, `apply-pr`, `exec`, and `apply-ps`.

As an example, we present the definition of the function `try-meta-apply-pr`.

```
ceq try-meta-apply-pr(Meta-Conf, Meta-Prog) =
          if Result?  ::  Result4Tuple
          then getTerm(Result?)
          else Meta-Conf
          fi
    if Result?  := metaXapply(Meta-Prog, Meta-Conf, 'apply-pr, ...)  .
```

The variable `Result?` is of sort `Result4Tuple?`. The function `metaXapply` takes the meta-representation of a module representing a 3APL program,[7] the meta-representation of a configuration, and the meta-representation of the label of the plan revision rewrite rule, i.e., `'apply-pr`, and yields the result of applying the plan revision rewrite rule to the configuration. If the rule application was successful, i.e., if `Result?` is of sort `Result4Tuple`, the term of the resulting 4-tuple which meta-represents the new configuration, is returned. Otherwise, the original unmodified configuration is returned. Note that there is only one object-level rule for plan revision (see Section 10.3.1), and that this is the one referred to in the definition of the function `try-meta-apply-pr`. Nevertheless, there might be multiple ways of applying this rule, since potentially multiple plan revision rules are applicable in a configuration. The function `metaXapply` then takes the first instance it finds.

A 3APL program can be executed through a deliberation cycle by calling Maude with the command.[8]

```
rewrite cycle(upTerm(conf),upModule('3APL-PROGRAM)) .
```

The function `cycle` uses the function `one-cycle` as specified above, to define the deliberation cycle as a Maude strategy. The term `conf` represents the initial configuration of the 3APL program, and `'3APL-PROGRAM` is the meta-representation of the name of the module containing the 3APL program.

10.4 Discussion and Related Work

10.4.1 Advantages of Maude

Based on our experience with the implementation of 3APL in Maude as elaborated on in Section 10.3, we argue that Maude is well suited as a prototyping

[7] The modules defining the syntax and semantics of 3APL are imported by this module, and are therefore also meta-represented.

[8] We omit some details for reasons of clarity.

and analysis tool for logic based cognitive agent programming languages.

In [Martí-Oliet and Meseguer, 2000], it is argued that rewriting logic is suitable both as a *logical* framework in which many other logics can be represented, and as a *semantic* framework.[9] The paper shows how to map Horn logic and linear logic in various ways to rewriting logic, and, among other things, it is observed that operational semantics can be naturally expressed in rewriting logic. The latter has been demonstrated from a more practical perspective in [Verdejo and Martí-Oliet, 2003], by demonstrating how simple functional, imperative, and concurrent languages can be implemented in Maude.

In this chapter, we show how (a simple version of) 3APL can be implemented in Maude. We observe that cognitive agent programming languages such as 3APL have a logical *as well as* a semantic component: the logical part consists of the belief and goal query languages (together with their respective satisfaction relations), and the semantic part consists of the transition system. Since Maude supports both the logical and the semantic component, the implementation of languages like 3APL in Maude is very natural, and the integration of the two components is seamless.

We observe that the direct mapping of transition rules of 3APL into rewrite rules of Maude ensures a *faithful* implementation of the operational semantics of 3APL in Maude. This direct mapping is a big advantage compared with the implementation of a 3APL interpreter in a general purpose language such as Java, in which the implementation is less direct. In particular, in Java one needs to program a mechanism for applying the specified transition rules in appropriate ways, whereas in the case of Maude the term rewriting engine takes care of this. As another approach of implementing a cognitive agent programming language in Java, one might consider to implement the plans of the agent as methods in Java, which is for example done in the Jadex framework [Pokahr et al., 2005b]. Since Java does not have support for revision of programs, implementing 3APL plans as methods in Java is not possible. We refer to Chapters 6, 7, and 8 for a theoretical treatment of the issues with respect to semantics of plan revision.

A faithful implementation of 3APL's semantics in Maude is very important with regard to our main original motivation for this work, i.e., to use the Maude LTL model checker to do formal verification for 3APL. The natural and transparent way in which the operational semantics of 3APL can be mapped to Maude, is a big advantage compared with the use of, e.g., the PROMELA language [Holzmann, 1991] in combination with the SPIN model checker [Holzmann, 1997].

SPIN is a generic verification system which supports the design and verification of asynchronous process systems. SPIN verification models are focused on proving the correctness of process interactions, and they attempt to abstract as much as possible from internal sequential computations. The language

[9] Obviously, logics often have semantics, but the notion of a *semantic framework* used in the cited paper refers to semantics of programming languages.

PROMELA is a high level language for specifying abstractions of distributed systems which can be used by SPIN, and it's main data structure is the message channel. In [Bordini et al., 2003], an implementation of the cognitive agent programming language AgentSpeak(F) - the finite state version of AgentSpeak(L) [Rao, 1996, Moreira and Bordini, 2002] - in PROMELA is described, for usage with SPIN. Most of the effort is devoted to translating AgentSpeak(F) into the PROMELA data structures. It is shown how to translate the data structures of AgentSpeak(F) into PROMELA channels. It is however not shown that this translation is correct, i.e., that the obtained PROMELA program correctly simulates the AgentSpeak(F) semantics. In contrast with the correctness of the implementation of 3APL in Maude, the correctness of the AgentSpeak(F) implementation in PROMELA is not obvious, because of the big gap between AgentSpeak(F) data structures and semantics, and PROMELA data structures and semantics. In [Eker et al., 2002], it is shown that the performance of the Maude model checker is comparable with that of SPIN with respect to a number of problems.

A further important advantage of Maude is that deliberation cycles can be programmed very naturally as strategies, using reflection. A related advantage is that a clear separation of the object-level and meta-level semantics can be maintained in Maude. A 3APL program can be executed without making use of a deliberation cycle, while it can equally easily be executed *with* a deliberation cycle.

10.4.2 Extending the Implementation

As was explained in Section 10.1, this chapter presents an implementation of a simplified version of 3APL in Maude. We however argue that the features of Maude are very well suited to support the implementation of various extensions of this version of 3APL. Implementing these extensions is left for future research.

In particular, an extension of a single-agent to a *multi-agent* version will be naturally implementable, since, from a computational point of view, rewriting logic is intrinsically concurrent [Martí-Oliet and Meseguer, 2000]. It was in fact the search for a general concurrency model that would help unify the heterogeneity of existing models, that provided the original impetus for the first investigations on rewriting logic [Meseguer, 1992].

Further, a more practically useful implementation will have to be *first-order*, rather than propositional. Although the implementation of a first-order version will be more involved, it can essentially be implemented in the same way as the current version, i.e., by mapping transition rules to rewrite rules. In [Dastani et al., 2004], the transition rules for a first-order version of 3APL are presented. Configurations in this setting have an extra substitution component, which records the assignment of values to variables. An implementation of this version in Maude will involve extending the notion of a configuration with such a substitution component, as specified in the cited paper.

Finally, we aim to extend the *logical part* in various ways, for which, as already pointed out, Maude is very well suited. Regarding this propositional version, one could think of extending the belief base and goal base to arbitrary sets of propositional formulas, rather than just sets of atoms. Also, the belief and goal query languages could be extended to query arbitrary propositional formulas. The satisfaction relations for queries could then be implemented using, e.g., tableau methods as suggested in [Martí-Oliet and Meseguer, 2000], for checking whether a propositional formula follows from the belief or goal base. Further, when considering a first-order version of 3APL, the belief base can be implemented as a set of Horn clauses, or even as a Prolog program. In the current Java implementation of 3APL, the belief base is implemented as a Prolog program. How to define standard Prolog in rewriting logic has been described in [Kulas and Beierle, 2000]. Finally, we aim to experiment with the implementation of more sophisticated specifications of the goals of 3APL agents and their accompanying satisfaction relations, such as proposed in Chapter 4.

10.4.3 Related Work

Besides the related work as already discussed in Sections 10.4.1 and 10.4.2, we mention a number of papers on Maude and agents. To the best of our knowledge, Maude has not been used widely in the agent community, and in particular not in the area of agent programming languages. Nevertheless, we found a small number of papers describing the usage of Maude in the agent systems field, which we will briefly discuss in this section.

A recent paper describes the usage of Maude for the specification of DIMA multi-agent models [Boudiaf et al., 2005]. In that paper, the previously not formalized DIMA model of agency is formalized using Maude. This work thus differs from our approach in that it does not implement an agent *programming language* which already has a formal semantics, independent of Maude. Consequently, its techniques for implementation are less principled and differing from ours.

Further, Maude has been used in the mobile agent area for checking fault-tolerant agent-based protocols used in the DaAgent system [Baalen et al., 2001]. Protocols in the DaAgent system are related to mobility issues, such as detection of node failure. The authors remark that the Java implementation for testing their protocols has proved to be "extremely time-consuming and inflexible". Using Maude, the protocol specifications are formalized and they can be debugged using the Maude model checker. Another example of the usage of Maude in the mobile agent area is presented in [Durán et al., 2000]. In that paper, Mobile Maude is presented, which is a mobile agent language extending Maude, and supporting mobile computation.

Chapter 11

Conclusion

"Semantics is a strange kind of applied mathematics; it seeks profound definitions rather than difficult theorems" [Karp et al., 1980, page 288].[1] This is a profound statement. It captures the essence of what semantics is all about.

When designing a programming language, the first step is to come up with a set of language constructs that should constitute the language. The choice of language constructs is motivated by the reason for designing the language. At this point, one generally has intuitions about the meaning of these constructs. The role of semantics is then to make these intuitions precise. This is where the process of carefully shaping the language begins.

Often, one will discover that the initial specification of the semantics does not fully capture the intuitions, or even yields counterintuitive or undesired behavior. Also, the semantics may turn out to be quite complex. This starts off a process of reconsideration of the semantic definitions, which might also result in changes to the syntax of the language.

Once the language definition has reached a point at which the language designer is reasonably satisfied with the result, the time comes to study the language, in order to get a better understanding of the proposed language constructs and their semantics. That is, semantic properties have to be investigated, and the language can be compared with other languages. Also, verification can be addressed. Finally, the language needs to be implemented, in order to test it in practice. All of these efforts may again result in changes to the syntax and semantics of the language.

This thesis has presented results concerning all these aspects of language design, i.e., the initial proposal for a language, studying properties of the language constructs, verification, and implementation.

[1] In [Tennent, 1991], this quote is attributed to Reynolds, who is a co-author of [Karp et al., 1980].

11.1 Part I: Goals

In Chapter 2, we have presented a simple cognitive agent programming language in which an agent has beliefs, goals, a plan, and plan selection and plan revision rules. In Chapter 3, we have studied a particular aspect of this language, i.e., we have studied the semantics of abstract plans (or achievement goals) in combination with plan revision rules. In particular, we have provided a language containing declarative subgoals, and have compared the semantics of abstract plans with the semantics of these subgoals. We have argued that the semantics of abstract plans interprets these in a procedural way. Nevertheless, we have shown that abstract plans can be programmed to behave as declarative subgoals, by proving a weak bisimulation between a language containing abstract plans, and a language containing declarative subgoals.

In Chapter 4, we have further investigated the representation of goals in cognitive agent programming languages. In particular, we have addressed the representation of conflicting goals, and (conflicting) goals that may be conditional on beliefs and other goals. We have proposed an alternative to the semantics of goal formulas of Chapter 2, that does not trivialize the logic in the case of an inconsistent goal base. That semantics was shown to be an extension of Hindriks' semantics [Hindriks et al., 2001], allowing the derivation of more goals (Proposition 4.2).

In order to allow the representation of conditional goals, we have proposed a language construct called goal adoption rules. It turned out that the semantics of goal formulas, given these goal adoption rules, could be defined using default logic. Default logic was designed to handle conflicting information, and it is also a natural way of handling conflicting goals.

In contrast with other approaches using default logic for the representation of goals, our approach defines the semantics of a full logical language of goal formulas. Further, our approach allows to express that certain goals are conflicting, even though they are logically consistent. This is achieved by translating negative goal formulas in the antecedent of goal adoption rules to the justifications of the corresponding default rules. We have investigated the proposed semantics by studying properties of the goal operator and comparing these with the axioms of modal logics. Also, we have investigated the relation between various semantics for goal formulas. Further, we have addressed the semantics of intention generation, as this is based on the semantics of goals. We have proposed a semantics for intention generation that is only partly satisfactory, and have identified a number of issues with respect to this semantics.

In Chapter 5, we have tried to put our research regarding goals in cognitive agent programming in perspective, by providing a broad overview of the various approaches that in some form address the incorporation of goals in agent programming frameworks. We have tried to provide some structure to the area by identifying important strands of research regarding ways in which goals have been represented and used in agent programming frameworks.

11.2 Part II: Plan Revision

In Part II, we have investigated the semantics of plans in the context of plan revision rules. The operational semantics of plans is not compositional, due to the fact that the heads of plan revision rules may contain arbitrary plans, rather than atomic plans. It is important that the semantics is compositional, in order to be able to define a compositional proof system.

In Chapter 6, we have addressed this issue by proposing a meta-language with a denotational semantics, which is compositional. We have shown how this meta-language is related to the object language using the operational semantics of the meta-language and the object language. In particular, we have provided a specific meta-program, the operational semantics of which was shown to be equivalent with the operational semantics of the object language. Further, we have shown that the denotational semantics of the meta-language is equivalent with the meta-level operational semantics.

In Chapter 7, we have addressed the issue of reasoning about plans in the context of plan revision rules by providing a dynamic logic that is tailored to handle plan revision. Because of the fact that the operational semantics of plans is not compositional in the context of plan revision, plans cannot be analyzed by structural induction. This means that standard propositional dynamic logic cannot be used to reason about these plans. Instead, we proposed a logic of restricted plans with sound and complete axiomatization. We also showed that this logic can be extended to a logic for non-restricted plans. This however results in an infinitary axiom system. We suggested that a possible way of dealing with the infinitary nature of the axiom system, is reasoning by induction on the restriction parameter. We showed some examples of how this could be done. Finally, we discussed the relation between plan revision rules and procedures.

The approach to the issue of compositionality of the semantics of plans that we take in Chapter 8, is to try to restrict the allowed plan revision rules,[2] such that the semantics of plans becomes compositional in some sense. It is not immediately obvious what kind of restriction would yield the desired result. We have proposed a restriction and proven that the semantics of plans in that case is compositional. Defining a proof system for these restricted plan revision rules is left for future research.

11.3 Part III: Software Engineering Aspects

In Chapter 9, we address the issue of modularization in cognitive agent programming languages. Modularization is widely recognized as a central issue in

[2]Note that in Chapter 8, we syntactically restrict *plan revision rules*, while in Chapter 7 we restrict the execution of *plans* by constraining the number of times that plan revision rules can be applied during execution of these plans.

software engineering, and has several advantages. The kind of modularization we have proposed is based on the goals of an agent, and was termed goal-oriented modularity. The idea is to incorporate modules into cognitive agent programming languages that encapsulate the information on how to achieve a goal, or a set of (related) goals. That is, modules contain information about the plans that can be used to achieve a (set of) goal(s). At run-time, the agent can then dispatch a goal to a module, which, broadly speaking, tries to achieve the dispatched goal using the information about plans contained in the module.

We have made the idea of goal-oriented modularity more concrete by proposing an extension to a cognitive agent programming language resembling the language of Chapter 2. This extended language contains modules consisting of plan selection and plan revision rules. Further, the plans are extended with a construct for calling a module with a certain (declarative) goal. The semantics of the extended language is provided.

In Chapter 10, we suggest the use of the Maude term rewriting language for prototyping cognitive agent programming languages such as the one provided in Chapter 2. We have observed that these cognitive agent programming languages have a logical as well as a semantic component: the logical part consists of the belief and goal query languages (together with their respective satisfaction relations), and the semantic part consists of the transition system. Since Maude supports both the logical and the semantic component, the implementation of such languages in Maude is very natural, and the integration of the two components is seamless. In particular, the translation of the transition rules of the transition system into rewrite rules of Maude is very direct, ensuring a faithful implementation of the semantics. Another advantage of Maude is that the language comes with an LTL model checker, which can be used for verifying cognitive agent programs.

11.4 Final Remarks

As was stated in [Karp et al., 1980], semantics seeks profound definitions rather than difficult theorems. All aspects of the process of designing a programming language, from an initial definition of the language to an investigation of its semantic properties to implementation and testing, in the end aim at defining a language that is somehow the "right" language for a certain purpose.

Proving that a language obeys certain desired properties provides some handle on the question of whether the language is the right language - or perhaps *a* right language. However, a set of properties never completely defines a language. Any language that emanates from the mind of a language designer is thus in large part a product of the intuitions of the designer. This is part of what makes semantics so difficult, and yet so interesting. As Wooldridge puts it in the foreword of [Bordini et al., 2005a, page xxix]: "[...] every programmer knows that what makes a "good" programming language is at least in part a

kind of magic: there is an indefinable "rightness" to the best languages, that make them somehow easier, more fun, more natural, just *better* to use".

In this thesis, we have proposed several language constructs with accompanying semantics that may be used in cognitive agent programming languages. Moreover, we have investigated properties of these and existing constructs. We hope that our efforts have resulted in a better understanding of the discussed constructs, but even more so we hope that the readers see in them some of that elusive "rightness", even if it is just a glimpse.

Bibliography

[Antoniou, 1997] Antoniou, G. (1997). *Nonmonotonic Reasoning*. Artificial Intelligence. The MIT Press, Cambridge, Massachusetts.

[Apt, 1981] Apt, K. R. (1981). Ten years of Hoare's logic: A survey - part I. *ACM Transactions of Programming Languages and Systems*, 3(4):431–483.

[Baalen et al., 2001] Baalen, J. V., Caldwell, J. L., and Mishra, S. (2001). Specifying and checking fault-tolerant agent-based protocols using Maude. In *FAABS '00: Proceedings of the First International Workshop on Formal Approaches to Agent-Based Systems-Revised Papers*, volume 1871 of *LNCS*, pages 180–193, London, UK. Springer-Verlag.

[Bellifemine et al., 2000] Bellifemine, F., Poggi, A., Rimassa, G., and Turci, P. (2000). An object oriented framework to realize agent systems. In *Proceedings of WOA 2000 Workshop*, pages 52–57. WOA.

[Bergstra et al., 1990] Bergstra, J. A., Heering, J., and Klint, P. (1990). Module algebra. *Journal of the Association for Computing Machinery*, 37(2):335–372.

[Besnard and Hunter, 1995] Besnard, P. and Hunter, A. (1995). Quasi-classical logic: Non-trivializable classical reasoning from incosistent information. In *Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 44–51.

[Blackburn et al., 2001] Blackburn, P., de Rijke, M., and Venema, Y. (2001). *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge.

[Bordini et al., 2005a] Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A. (2005a). *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin.

[Bordini et al., 2003] Bordini, R. H., Fisher, M., Pardavila, C., and Wooldridge, M. (2003). Model checking AgentSpeak. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 409–416, Melbourne.

[Bordini et al., 2005b] Bordini, R. H., Hübner, J. F., and Vieira, R. (2005b). Jason and the golden fleece of agent-oriented programming. In Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin.

[Bordini and Moreira, 2004] Bordini, R. H. and Moreira, Á. F. (2004). Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1–3):197–226. Special Issue on Computational Logic in Multi-Agent Systems.

[Boudiaf et al., 2005] Boudiaf, N., Mokhati, F., Badri, M., and Badri, L. (2005). Specifying DIMA multi-agent models using Maude. In *Intelligent Agents and Multi-Agent Systems, 7th Pacific Rim International Workshop on Multi-Agents (PRIMA 2004)*, volume 3371 of *LNCS*, pages 29–42. Springer, Berlin.

[Bratman, 1987] Bratman, M. E. (1987). *Intention, plans, and practical reason*. Harvard University Press, Massachusetts.

[Braubach et al., 2006] Braubach, L., Pokahr, A., and Lamersdorf, W. (2006). Extending the capability concept for flexible BDI agent modularization. In Bordini, R. H., Dastani, M., Dix, J., and Seghrouchni, A. E. F., editors, *Proceedings of the Third International Workshop on Programming Multiagent Systems (ProMAS'05)*, volume 3862 of *LNAI*, pages 139–155. Springer-Verlag.

[Braubach et al., 2005] Braubach, L., Pokahr, A., Moldt, D., and Lamersdorf, W. (2005). Goal representation for BDI agent systems. In *Programming multiagent systems, second international workshop (ProMAS'04)*, volume 3346 of *LNAI*, pages 44–65. Springer, Berlin.

[Bresciani et al., 2004] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., and Perini, A. (2004). Tropos: An agent-oriented software development methodology. *Journal of Autonomous Agents and Multi-Agent Systems*, 8(3):203–236.

[Brewka, 1991] Brewka, G. (1991). *Nonmonotonic reasoning: logical foundations of commonsense*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge.

[Brewka et al., 1997] Brewka, G., Dix, J., and Konolige, K. (1997). *Nonmonotonic reasoning: an overview*. CSLI Publications, Stanford.

[Broersen et al., 2002] Broersen, J., Dastani, M., Hulstijn, J., and van der Torre, L. (2002). Goal generation in the BOID architecture. *Cognitive Science Quarterly*, 2(3-4):428–447.

[Busetta et al., 2000] Busetta, P., Howden, N., Rönnquist, R., and Hodgson, A. (2000). Structuring BDI agents in functional clusters. In *ATAL '99: 6th International Workshop on Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL),*, pages 277–289, London, UK. Springer-Verlag.

[Castañeda, 1981] Castañeda, H.-N. (1981). The paradoxes of deontic logic. The simplest solution to all of them in one fell swoop. *New studies in deontic logic: norms, actions and the foundations of ethics*, pages 37–85.

[Chellas, 1980] Chellas, B. F. (1980). *Modal Logic: An Introduction*. Cambridge University Press, Cambridge.

[Clavel et al., 2005] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. (2005). Maude manual (version 2.1.1).

[Clavel and Meseguer, 1996] Clavel, M. and Meseguer, J. (1996). Reflection and strategies in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 4:125–147.

[Cohen and Levesque, 1990] Cohen, P. R. and Levesque, H. J. (1990). Intention is choice with commitment. *Artificial Intelligence*, 42:213–261.

[Dastani et al., 2003] Dastani, M., de Boer, F. S., Dignum, F., and Meyer, J.-J. Ch. (2003). Programming agent deliberation – an approach illustrated using the 3APL language. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 97–104, Melbourne.

[Dastani et al., 2005a] Dastani, M., Governatori, G., Rotolo, A., and van der Torre, L. (2005a). Programming cognitive agents in defeasible logic. In *Proceedings of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05)*, volume 3835 of *LNAI*, pages 621–637. Springer-Verlag.

[Dastani and van der Torre, 2004] Dastani, M. and van der Torre, L. (2004). Programming BOID-Plan agents: deliberating about conflicts among defeasible mental attitudes and plans. In *Proceedings of the Third Conference on Autonomous Agents and Multi-agent Systems (AAMAS'04)*, pages 706–713, New York, USA.

[Dastani et al., 2004] Dastani, M., van Riemsdijk, M. B., Dignum, F., and Meyer, J.-J. Ch. (2004). A programming language for cognitive agents: goal directed 3APL. In *Programming multiagent systems, first international workshop (ProMAS'03)*, volume 3067 of *LNAI*, pages 111–130. Springer, Berlin.

[Dastani et al., 2005b] Dastani, M., van Riemsdijk, M. B., Hulstijn, J., Dignum, F., and Meyer, J.-J. Ch. (2005b). Enacting and deacting roles in agent programming. In Odell, J., Giorgini, P., and Müller, J., editors, *Agent-Oriented*

Software Engineering V, volume 3382 of *LNCS*, pages 189–204. Springer-Verlag.

[Dastani et al., 2005c] Dastani, M., van Riemsdijk, M. B., and Meyer, J.-J. Ch. (2005c). Programming multi-agent systems in 3APL. In Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin.

[Dastani et al., 2006] Dastani, M., van Riemsdijk, M. B., and Meyer, J.-J. Ch. (2006). Goal types in agent programming. In *Proceedings of the 17th European Conference on Artifical Intelligence 2006 (ECAI'06)*. To appear.

[de Bakker, 1980] de Bakker, J. (1980). *Mathematical Theory of Program Correctness*. Series in Computer Science. Prentice-Hall International, London.

[Dennett, 1987] Dennett, D. (1987). *The Intentional Stance*. MIT Press, Cambridge MA.

[Dignum and Conte, 1997] Dignum, F. and Conte, R. (1997). Intentional agents and goal formation. In *Agent Theories, Architectures, and Languages*, pages 231–243.

[d'Inverno et al., 1998] d'Inverno, M., Kinny, D., Luck, M., and Wooldridge, M. (1998). A formal specification of dMARS. In *ATAL '97: Proceedings of the 4th International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages*, pages 155–176, London, UK. Springer-Verlag.

[Dix and Zhang, 2005] Dix, J. and Zhang, Y. (2005). IMPACT: a multi-agent framework with declarative semantics. In Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin.

[Douglas and Saunders, 2003] Douglas, G. and Saunders, S. (2003). The Philosopher's Magazine Online: Philosopher of the Month, April 2003, Dan Dennett. `http://www.philosophers.co.uk/cafe/phil_apr2003.htm`.

[Drabble et al., 1997] Drabble, B., Dalton, J., and Tate, A. (1997). Repairing plans on the fly. In *Proceedings of the NASA Workshop on Planning and Scheduling for Space*.

[Durán et al., 2000] Durán, F., Eker, S., Lincoln, P., and Meseguer, J. (2000). Principles of mobile Maude. In *ASA/MA 2000: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, volume 1882 of *LNCS*, pages 73–85, London, UK. Springer-Verlag.

[Egli, 1975] Egli, H. (1975). A mathematical model for nondeterministic computations. Technical report, ETH, Zürich.

[Eker et al., 2002] Eker, S., Meseguer, J., and Sridharanarayanan, A. (2002). The Maude LTL model checker. In Gaducci, F. and Montanari, U., editors, *Proceedings of the 4th International Workshop on Rewriting Logic and Its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier.

[E.M.Clarke et al., 2000] E.M.Clarke, Grumberg, O., and Peled, D. (2000). *Model Checking*. MIT Press.

[Evertsz et al., 2004] Evertsz, R., Fletcher, M., Jones, R., Jarvis, J., Brusey, J., and Dance, S. (2004). Implementing industrial multi-agent systems using JACK™. In *Proceedings of the first international workshop on programming multiagent systems (ProMAS'03)*, volume 3067 of *LNAI*, pages 18–49. Springer, Berlin.

[Fikes and Nilsson, 1971] Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208.

[Fisher, 1997] Fisher, M. (1997). Implementing BDI-like systems by direct execution. In *Proceedings of the International Joint Conference on AI (IJCAI'97)*, pages 316–321. Morgan-Kaufmann.

[Fisher, 2006] Fisher, M. (2006). METATEM: The story so far. In Bordini, R. H., Dastani, M., Dix, J., and Seghrouchni, A. E. F., editors, *Proceedings of the Third International Workshop on Programming Multiagent Systems (ProMAS'05)*, volume 3862 of *LNAI*, pages 3–22. Springer-Verlag.

[Gabbay and Hunter, 1991] Gabbay, D. and Hunter, A. (1991). Making inconsistency respectable: A logical framework for inconsistency in reasoning. In Jorrand, P. and Kelemen, J., editors, *Proceedings of Fundamentals of Artifical Intelligence Research (FAIR'91)*, pages 19–32. Springer-Verlag.

[Georgeff and Lansky, 1987] Georgeff, M. and Lansky, A. (1987). Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, pages 677–682.

[Georgeff et al., 1999] Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M. (1999). The Belief-Desire-Intention model of agency. In Muller, J., Singh, M., and Rao, A., editors, *Intelligent Agents V (ATAL'98)*, volume 1365 of *LNAI*. Springer-Verlag.

[Ghezzi et al., 1991] Ghezzi, C., Jazayeri, M., and Mandrioli, D. (1991). *Fundamentals of software engineering*. Prentice-Hall International, London.

[Giacomo et al., 2000] Giacomo, G. d., Lespérance, Y., and Levesque, H. (2000). *ConGolog*, a Concurrent Programming Language Based on the Situation Calculus. *Artificial Intelligence*, 121(1-2):109–169.

[Giacomo and Levesque, 1999] Giacomo, G. D. and Levesque, H. J. (1999). An incremental interpreter for high-level programs with sensing. In Levesque, H. J. and Pirri, F., editors, *Logical foundations for cognitive agents*, pages 86–102. Springer-Verlag.

[Gosling et al., 2000] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2000). *The Java Language Specification*. Addison-Wesley, second edition.

[Governatori and Rotolo, 2004] Governatori, G. and Rotolo, A. (2004). Defeasible logic: Agency, intention and obligation. In Lomuscio, A. and Nute, D., editors, *Deontic Logic in Computer Science (DEON'04)*, volume 3065 of *LNAI*, pages 114–128. Springer, Berlin.

[Hammond, 1990] Hammond, K. J. (1990). Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228.

[Hansson, 1969] Hansson, B. (1969). An analysis of some deontic logics. In *Nous 3*.

[Harel, 1979] Harel, D. (1979). *First-Order Dynamic Logic*. Lectures Notes in Computer Science 68. Springer, Berlin.

[Harel et al., 2000] Harel, D., Kozen, D., and Tiuryn, J. (2000). *Dynamic Logic*. The MIT Press, Cambridge, Massachusetts and London, England.

[Harrenstein, 2004] Harrenstein, B. P. (2004). *Logic in Conflict: Logical Explorations in Strategic Equilibrium*. PhD thesis.

[Hindriks et al., 1999a] Hindriks, K., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. Ch. (1999a). Control structures of rule-based agent languages. In Müller, J., Singh, M. P., and Rao, A. S., editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 381–396. Springer-Verlag: Heidelberg, Germany.

[Hindriks, 2001] Hindriks, K. V. (2001). *Agent programming languages - programming with mental models*. PhD thesis.

[Hindriks et al., 1998] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. Ch. (1998). A formal embedding of AgentSpeak(L) in 3APL. In Antoniou, G. and Slaney, J., editors, *Advanced Topics in Artificial Intelligence*, pages 155–166. Springer, LNAI 1502.

[Hindriks et al., 1999b] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. Ch. (1999b). Agent programming in 3APL. *Int. J. of Autonomous Agents and Multi-Agent Systems*, 2(4):357–401.

[Hindriks et al., 2000] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. Ch. (2000). A programming logic for part of the agent language 3APL. In *Proceedings of the First Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS'00)*.

[Hindriks et al., 2001] Hindriks, K. V., de Boer, F. S., van der Hoek, W., and Meyer, J.-J. Ch. (2001). Agent programming with declarative goals. In *Intelligent Agents VI - Proceedings of the 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL'2000)*, Lecture Notes in AI. Springer, Berlin.

[Hindriks et al., 2002] Hindriks, K. V., Lespirance, Y., and Levesque, H. (2002). A formal embedding of ConGolog in 3APL. In *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 558–562.

[Holzmann, 1991] Holzmann, G. (1991). *Design and Validation of Computer Protocols*. Prentice Hall, New Jersey.

[Holzmann, 1997] Holzmann, G. (1997). The model checker SPIN. *IEEE Trans. Software Engineering*, 23(5):279–295.

[Horty, 1993] Horty, J. F. (1993). Deontic logic as founded on nonmonotonic logic. *Annals of Mathematics and Artificial Intelligence (Special Issue on Deontic Logic in Computer Science)*, 9:69–91.

[Horty, 1994] Horty, J. F. (1994). Moral dilemmas and nonmonotonic logic. *Journal of Philosophical Logic*, 23(1):35–65.

[Horty, 1997] Horty, J. F. (1997). Nonmonotonic foundations for deontic logic. In Nute, D., editor, *Defeasible Deontic Logic*, pages 17–44. Kluwer Academic Publishers.

[Huber, 1999] Huber, M. J. (1999). JAM: A BDI-theoretic mobile agent architecture. In *Proceedings of the third international conference on autonomous agents (Agents'99)*, pages 236–243.

[Hübner et al., 2006] Hübner, J. F., Bordini, R. H., and Wooldridge, M. (2006). Declarative goal patterns for AgentSpeak. In *Proceedings of the fourth International Workshop on Declarative Agent Languages and Technologies (DALT'06)*.

[Ingrand et al., 1992] Ingrand, F. F., Georgeff, M. P., and Rao, A. S. (1992). An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34–44.

[Jennings, 1999] Jennings, N. (1999). Agent-oriented software engineering. In *Proceedings of the 12th International Conference on Industrial and Engineering Applications of AI*, pages 4–10. Invited paper.

[Jennings et al., 1996] Jennings, N., Faratin, P., Johnson, M. J., Norman, T. J., O'Brien, P., and Wiegand, M. E. (1996). Agent-based business process management. *International Journal of Cooperative Information Systems*, 5(2–3):105–130.

[Karp et al., 1980] Karp, R. M., Manna, Z., Meyer, A. R., Reynolds, J. C., Ritchie, R. W., Ullman, J. D., and Winograd, S. (1980). Theory of computation. In Arden, B. W., editor, *What Can Be Automated? The Computer Science and Engineering Research Study*, pages 137–295. The MIT Press.

[Kuiper, 1981] Kuiper, R. (1981). An operational semantics for bounded nondeterminism equivalent to a denotational one. In de Bakker, J. W. and van Vliet, J. C., editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 373–398. North-Holland.

[Kulas and Beierle, 2000] Kulas, M. and Beierle, C. (2000). Defining standard Prolog in rewriting logic. In *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier Science Publishers.

[Maher and Governatori, 1999] Maher, M. J. and Governatori, G. (1999). A semantic decomposition of defeasible logics. In *Proceedings of the American National Conference on Artificial Intelligence (AAAI'99)*, pages 299–305.

[Martí-Oliet and Meseguer, 2000] Martí-Oliet, N. and Meseguer, J. (2000). Rewriting logic as a logical and semantic framework. In Meseguer, J., editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers.

[Meseguer, 1992] Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155.

[Meseguer, 1997] Meseguer, J. (1997). Membership algebra as a logical framework for equational specification. In *WADT '97: Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 18–61, London, UK. Springer-Verlag.

[Meyer, 1988] Meyer, B. (1988). *Object-oriented software construction*. Series in Computer Science. Prentice-Hall International, London.

[Meyer and van der Hoek, 1995] Meyer, J.-J. Ch. and van der Hoek, W. (1995). *Epistemic logic for AI and computer science*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge.

[Moreira and Bordini, 2002] Moreira, A. and Bordini, R. (2002). An operational semantics for a BDI agent-oriented programming language. In *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS'02)*.

[Moreira et al., 2004] Moreira, A. F., Vieira, R., and Bordini, R. H. (2004). Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. In *Declarative Agent Languages and Technologies, First International Workshop (DALT'03)*, volume 2990 of *LNAI*, pages 135–154, London, UK. Springer-Verlag.

[Mosses, 1990] Mosses, P. D. (1990). Denotational semantics. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 575–631. Elsevier, Amsterdam.

[Nebel and Koehler, 1995] Nebel, B. and Koehler, J. (1995). Plan reuse versus plan generation: a theoretical and empirical analysis. *Artificial Intelligence*, 76:427–454.

[Norling, 2003] Norling, E. (2003). Capturing the quake player: using a BDI agent to model human behaviour. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03), poster*, pages 1080–1081, Melbourne.

[Nute, 1994] Nute, D. (1994). Defeasible logic. 3:353–395.

[Ölveczky, 2005] Ölveczky, P. C. (2005). Formal modeling and analysis of distributed systems in Maude. Lecture Notes.

[Plotkin, 1981] Plotkin, G. D. (1981). A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus.

[Pokahr et al., 2005a] Pokahr, A., Braubach, L., and Lamersdorf, W. (2005a). A goal deliberation strategy for BDI agent systems. In *MATES 2005*, volume 3550 of *LNAI*, pages 82–93. Springer-Verlag.

[Pokahr et al., 2005b] Pokahr, A., Braubach, L., and Lamersdorf, W. (2005b). Jadex: a BDI reasoning engine. In Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin.

[Poole, 1988] Poole, D. (1988). A logical framework for default reasoning. *Artificial Intelligence*, 36:27–47.

[Rao, 1996] Rao, A. S. (1996). AgentSpeak(L): BDI agents speak out in a logical computable language. In van der Velde, W. and Perram, J., editors, *Agents Breaking Away (LNAI 1038)*, pages 42–55. Springer-Verlag.

[Rao and Georgeff, 1991] Rao, A. S. and Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. In Allen, J., Fikes, R., and Sandewall, E., editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann.

[Rao and Georgeff, 1998] Rao, A. S. and Georgeff, M. P. (1998). Decision procedures for BDI logics. *Journal of Logic and Computation*, 8(3):293.

[Reiter, 1980] Reiter, R. (1980). A logic for default-reasoning. *Artificial Intelligence*, 13:81–132.

[Reiter, 1987] Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95.

[Sardina and Shapiro, 2003] Sardina, S. and Shapiro, S. (2003). Rational action in agent programs with prioritized goals. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 417–424, Melbourne.

[Segerberg, 1989] Segerberg, K. (1989). Bringing it about. *Journal of Philosophical Logic*, 18:327–347.

[Shoham, 1993] Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence*, 60:51–92.

[Simon et al., 2006] Simon, G., Mermet, B., and Fournier, D. (2006). Goal decomposition tree: An agent model to generate a validated agent behaviour. In *Declarative Agent Languages and Technologies III, Third International Workshop (DALT'05)*, volume 3904 of *LNAI*, pages 124–140, London, UK. Springer-Verlag.

[Stoy, 1977] Stoy, J. E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA.

[Tennent, 1991] Tennent, R. (1991). *Semantics of Programming Languages*. Series in Computer Science. Prentice-Hall International, London.

[Thangarajah et al., 2003a] Thangarajah, J., Padgham, L., and Winikoff, M. (2003a). Detecting and avoiding interference between goals in intelligent agents. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*.

[Thangarajah et al., 2003b] Thangarajah, J., Padgham, L., and Winikoff, M. (2003b). Detecting and exploiting positive goal interaction in intelligent agents. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 401–408, Melbourne.

[Thangarajah et al., 2002] Thangarajah, J., Winikoff, M., Padgham, L., and Fischer, K. (2002). Avoiding resource conflicts in intelligent agents. In van Harmelen, F., editor, *Proceedings of the 15th European Conference on Artifical Intelligence 2002 (ECAI 2002)*, Lyon, France.

[Thomason, 2000] Thomason, R. H. (2000). Desires and defaults: A framework for planning with inferred goals. In Cohn, A. G., Giunchiglia, F., and Selman, B., editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 702–713, San Francisco. Morgan Kaufmann.

[van der Hoek et al., 1998] van der Hoek, W., van Linder, B., and Meyer, J.-J. Ch. (1998). An integrated modal approach to rational agents. In Wooldridge, M. and Rao, A. S., editors, *Foundations of Rational Agency*, Applied Logic Series 14, pages 133–168. Kluwer, Dordrecht.

[van der Hoek and Wooldridge, 2003] van der Hoek, W. and Wooldridge, M. (2003). Towards a logic of rational agency. *Logic Journal of the IGPL*, 11(2):133.

[van der Krogt and de Weerdt, 2005a] van der Krogt, R. P. and de Weerdt, M. M. (2005a). Plan repair as an extension of planning. In *Proceedings of the International Conference on Planning and Scheduling (ICAPS'05)*, pages 161–170.

[van der Krogt and de Weerdt, 2005b] van der Krogt, R. P. and de Weerdt, M. M. (2005b). Plan repair using a plan library. In *Proceedings of the Belgium-Dutch Conference on Artificial Intelligence (BNAIC'05)*, pages 254–259. BNVKI.

[van Emde Boas, 1978] van Emde Boas, P. (1978). The connection between modal logic and algorithmic logics. In *Mathematical foundations of computer science 1978*, volume 64 of *LNCS*, pages 1–15. Springer, Berlin.

[van Fraassen, 1973] van Fraassen, B. C. (1973). Values and the heart's command. *Journal of Philosophy*, 70(1):5–19.

[van Lamsweerde and Letier, 2004] van Lamsweerde, A. and Letier, E. (2004). From object orientation to goal orientation: a paradigm shift for requirements engineering. In *Radical Innovations of Software and Systems Engineering in the Future: 9th International Workshop (RISSEF'02)*, volume 2941 of *LNCS*, pages 325–340, London, UK. Springer-Verlag.

[van Riemsdijk, 2002] van Riemsdijk, M. B. (2002). Agent programming in Dribble: from beliefs to goals with plans. Master's thesis, Utrecht University.

[van Riemsdijk et al., 2005a] van Riemsdijk, M. B., Dastani, M., Dignum, F., and Meyer, J.-J. Ch. (2005a). Dynamics of declarative goals in agent programming. In Leite, J. A., Omicini, A., Torroni, P., and Yolum, P., editors, *Declarative agent languages and technologies II: second international workshop (DALT'04)*, volume 3476 of *LNAI*, pages 1–18.

[van Riemsdijk et al., 2005b] van Riemsdijk, M. B., Dastani, M., and Meyer, J.-J. Ch. (2005b). Semantics of declarative goals in agent programming. In *Proceedings of the fourth international joint conference on autonomous agents and multiagent systems (AAMAS'05)*, pages 133–140, Utrecht.

[van Riemsdijk et al., 2005c] van Riemsdijk, M. B., Dastani, M., and Meyer, J.-J. Ch. (2005c). Subgoal semantics in agent programming. In Bento, C., Cardoso, A., and Dias, G., editors, *Progress in Artifical Intelligence: 12th Portuguese Conference on Artificial Intelligence (EPIA'05)*, volume 3808 of *LNAI*, pages 548–559. Springer-Verlag.

[van Riemsdijk et al., 2006a] van Riemsdijk, M. B., Dastani, M., Meyer, J.-J. Ch., and de Boer, F. S. (2006a). Goal-oriented modularity in agent programming. In *Proceedings of the fifth international joint conference on autonomous agents and multiagent systems (AAMAS'06)*, pages 1271–1278, Hakodate.

[van Riemsdijk et al., 2006b] van Riemsdijk, M. B., de Boer, F. S., Dastani, M., and Meyer, J.-J. Ch. (2006b). Prototyping 3APL in the Maude term rewriting language. In *Proceedings of the Seventh Workshop on Computational Logic in Multi-Agent Systems (CLIMA'06)*. To appear in LNAI.

[van Riemsdijk et al., 2005d] van Riemsdijk, M. B., de Boer, F. S., and Meyer, J.-J. Ch. (2005d). Dynamic logic for plan revision in intelligent agents. In Leite, J. A. and Torroni, P., editors, *Computational logic in multi-agent systems: fifth international workshop (CLIMA'04)*, volume 3487 of *LNAI*, pages 16–32.

[van Riemsdijk et al., 2005e] van Riemsdijk, M. B., de Boer, F. S., and Meyer, J.-J. Ch. (2005e). Dynamic logic for plan revision in intelligent agents. Technical Report UU-CS-2005-013, Utrecht University, Institute of Information and Computing Sciences. To appear in Journal of Logic and Computation.

[van Riemsdijk and Meyer, 2006] van Riemsdijk, M. B. and Meyer, J.-J. Ch. (2006). A compositional semantics of plan revision in intelligent agents. In Johnson, M. and Vene, V., editors, *Algebraic Methodology And Software Technology: 11th International Conference, AMAST 2006*, volume 4019 of *LNCS*, pages 353–367. Springer-Verlag.

[van Riemsdijk et al., 2003a] van Riemsdijk, M. B., Meyer, J.-J. Ch., and de Boer, F. S. (2003a). Semantics of plan revision in intelligent agents. Technical report, Utrecht University, Institute of Information and Computing Sciences. UU-CS-2004-002.

[van Riemsdijk et al., 2004] van Riemsdijk, M. B., Meyer, J.-J. Ch., and de Boer, F. S. (2004). Semantics of plan revision in intelligent agents. In Rattray, C., Maharaj, S., and Shankland, C., editors, *Proceedings of the 10th*

International Conference on Algebraic Methodology And Software Technology (AMAST04), volume 3116 of *LNCS*, pages 426–442. Springer-Verlag.

[van Riemsdijk et al., 2006c] van Riemsdijk, M. B., Meyer, J.-J. Ch., and de Boer, F. S. (2006c). Semantics of plan revision in intelligent agents. *Theoretical Computer Science*, 351(2):240–257. Special issue of Algebraic Methodology and Software Technology (AMAST'04).

[van Riemsdijk et al., 2003b] van Riemsdijk, M. B., van der Hoek, W., and Meyer, J.-J. Ch. (2003b). Agent programming in Dribble: from beliefs to goals using plans. In *Proceedings of the second international joint conference on autonomous agents and multiagent systems (AAMAS'03)*, pages 393–400, Melbourne.

[Verdejo and Martí-Oliet, 2003] Verdejo, A. and Martí-Oliet, N. (2003). Executable structural operational semantics in Maude. Technical report, Universidad Complutense de Madrid, Madrid.

[Visser and Burkhard, 2006] Visser, U. and Burkhard, H.-D. (2006). RoboCup 2006, Bremen, Germany. `http://www.robocup2006.org`.

[von Wright, 1951] von Wright, G. H. (1951). Deontic logic. *Mind*, 60:1–15.

[Winikoff, 2005] Winikoff, M. (2005). JACK™ intelligent agents: an industrial strength platform. In Bordini, R. H., Dastani, M., Dix, J., and El Fallah Seghrouchni, A., editors, *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, Berlin.

[Winikoff et al., 2002] Winikoff, M., Padgham, L., Harland, J., and Thangarajah, J. (2002). Declarative and procedural goals in intelligent agent systems. In *Proceedings of the eighth international conference on principles of knowledge respresentation and reasoning (KR2002)*, Toulouse.

[Wirsing, 1990] Wirsing, M. (1990). Algebraic specification. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 675–788. Elsevier, Amsterdam.

[Wooldridge, 1997] Wooldridge, M. (1997). Agent-based software engineering. *IEEE Proceedings Software Engineering*, 144(1):26–37.

[Wooldridge, 2002] Wooldridge, M. (2002). *An introduction to multiagent systems*. John Wiley and Sons, LTD, West Sussex.

[Wooldridge and Ciancarini, 2001] Wooldridge, M. and Ciancarini, P. (2001). Agent-Oriented Software Engineering: The State of the Art. In Ciancarini, P. and Wooldridge, M., editors, *First Int. Workshop on Agent-Oriented Software Engineering*, volume 1957, pages 1–28. Springer-Verlag, Berlin.

[Wooldridge and Jennings, 1995] Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152.

[Yoshimura et al., 2000] Yoshimura, K., Rönnquist, R., and Sonenberg, L. (2000). An approach to specifying coordinated agent behaviour. In *PRIMA'00*, volume 1881 of *LNAI*, pages 115–127. Springer, Berlin.

[Zambonelli et al., 2003] Zambonelli, F., Jennings, N. R., and Wooldridge, M. (2003). Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3):317–370.

Programmeren van Cognitieve Agenten

Dit proefschrift gaat over het ontwerpen en onderzoeken van gespecialiseerde programmeertalen voor het programmeren van agenten. Wij concentreren ons hierbij in het bijzonder op programmeertalen voor rationele agenten. Onder rationele agenten verstaan wij stukjes software die zich flexibel gedragen en die in staat zijn om "goede" beslissingen te nemen over wat te doen.

Een belangrijke onderzoekslijn op dit gebied is gebaseerd op Bratman's zogenaamde *Belief Desire Intention* (BDI) filosofie. Het idee van de BDI filosofie is dat het gedrag van rationele agenten voorspeld en verklaard kan worden door aan deze agenten geloof (*beliefs*), wensen (*desires*) en intenties (*intentions*) toe te schrijven, en door aan te nemen dat de agent geneigd is om actie te ondernemen om zijn wensen te vervullen, waarbij hij rekening houdt met zijn geloof over de wereld.

Na de introductie van BDI filosofie, werd het idee geopperd dat het wellicht niet alleen mogelijk zou zijn het gedrag van rationele agenten te verklaren en te beschrijven in termen van de BDI begrippen, maar dat het misschien ook mogelijk zou zijn om rationele agenten te programmeren, waarbij de begrippen *beliefs*, *desires* en *intentions* als basis voor een programmeertaal gebruikt zouden worden. Het onderzoek dat in deze richting wordt uitgevoerd, gebruikt niet alleen de begrippen *beliefs*, *desires* en *intentions*, maar ook gerelateerde noties zoals *goals* (doelen) en *plans* (plannen). Al dit soort begrippen noemen wij "cognitieve" begrippen. Programmeertalen voor agenten die gebaseerd zijn op deze begrippen noemen wij "cognitieve-agentprogrammeertalen".

In ons werk stellen wij nieuwe programmeerconstructen voor om deze cognitieve begrippen in een programmeertaal te kunnen representeren, en wij onderzoeken bestaande constructen. Wij nemen een semantische benadering, in de zin dat wij formele semantiek definieren voor de voorgestele constructen, en wij onderzoeken de constructen door het doen van een semantische analyse. Wij onderzoeken in het bijzonder manieren voor het representeren van doelen, en wij bestuderen een construct genaamd "planrevisieregel" van de cognitieve-agentprogrammeertaal 3APL. Planrevisieregels kunnen gebruikt worden om het

plan van een agent te wijzigen als de omstandigheden daarom vragen.

Wat betreft het representeren van doelen onderzoeken wij in het bijzonder de representatie van subdoelen in de plannen van agenten. Wij laten zien hoe declaratieve subdoelen, dat wil zeggen subdoelen die een gewenste toestand aanduiden, kunnen worden geprogrammeerd in 3APL. Dit ondanks het feit dat de semantiek van 3APL zo gedefinieerd is dat subdoelen zich procedureel gedragen. Verder stellen wij een semantiek voor het representeren van conflicterende doelen voor die is gebaseerd op *default logic*, en wij onderzoeken eigenschappen van deze semantiek. Ook geven wij een analyse van manieren waarop doelen gerepresenteerd worden in bestaande cognitieve-agentprogrammeertalen.

Wat betreft planrevisieregels analyseren wij de semantische problemen die ontstaan bij het introduceren van deze regels. Dat wil zeggen, de semantiek van het executeren van plannen wordt niet-compositioneel met de introductie van deze regels. Dit is problematisch wanneer men wil redeneren over het uitvoeren van plannen. Wij stellen een dynamische logica voor die erop gericht is om met planrevisie om te gaan, door het probleem van niet-compositionaliteit op een bepaalde manier te omzeilen. Bovendien laten we zien hoe planrevisieregels beperkt kunnen worden, zodanig dat de semantiek weer compositioneel wordt.

Tot slot doen wij enkele voorstellen op het gebied van *software engineering*. Wij stellen een manier voor om ondersteuning voor modularisatie in cognitieve-agentprogrammeertalen te introduceren die gebaseerd is op de doelen van de agent. Bovendien laten we zien dat de Maude termherschrijftaal geschikt is voor het implementeren van logica-gebaseerde cognitieve-agentprogrammeertalen zoals 3APL.

Curriculum Vitae

Maria Birna van Riemsdijk

29 oktober 1978
Geboren te Wageningen.

september 1991 - augustus 1997
Voorbereidend Wetenschappelijk Onderwijs aan de Regionale Scholengemeenschap Pantarijn te Wageningen. Diploma behaald in juni 1997.

september 1997 - augustus 2002
Studie Informatica aan het Institute of Information and Computing Sciences van de Universiteit Utrecht. Diploma behaald in augustus 2002 (cum laude).

september 2002 - augustus 2006
Assistent in Opleiding aan het Institute of Information and Computing Sciences van de Universiteit Utrecht.

SIKS Dissertation Series

1998

Johan van den Akker, *DEGAS - An Active, Temporal Database of Autonomous Objects*, CWI, 1998-1

Floris Wiesman, *Information Retrieval by Graphically Browsing Meta-Information*, UM, 1998-2

Ans Steuten, *A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective*, TUD, 1998-3

Dennis Breuker, *Memory versus Search in Games*, UM, 1998-4

E.W. Oskamp, *Computerondersteuning bij Straftoemeting*, RUL, 1998-5

1999

Mark Sloof, *Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products*, VU, 1999-1

Rob Potharst, *Classification using decision trees and neural nets*, EUR, 1999-2

Don Beal, *The Nature of Minimax Search*, UM, 1999-3

Jacques Penders, *The practical Art of Moving Physical Objects*, UM, 1999-4

Aldo de Moor, *Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems*, KUB, 1999-5

Niek J.E. Wijngaards, *Re-design of compositional systems*, VU, 1999-6

David Spelt, *Verification support for object database design*, UT, 1999-7

Jacques H.J. Lenting, *Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.*, UM, 1999-8

2000

Frank Niessink, *Perspectives on Improving Software Maintenance*, VU, 2000-1

Koen Holtman, *Prototyping of CMS Storage Management*, TUE, 2000-2

Carolien M.T. Metselaar, *Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief*, UVA, 2000-3

Geert de Haan, *ETAG, A Formal Model of Competence Knowledge for User Interface Design*, VU, 2000-4

Ruud van der Pol, *Knowledge-based Query Formulation in Information Retrieval*, UM, 2000-5

Rogier van Eijk, *Programming Languages for Agent Communication*, UU, 2000-6

Niels Peek, *Decision-theoretic Planning of Clinical Patient Management*, UU, 2000-7

Veerle Coup, *Sensitivity Analyis of Decision-Theoretic Networks*, EUR, 2000-8

Florian Waas, *Principles of Probabilistic Query Optimization*, CWI, 2000-9

Niels Nes, *Image Database Management System Design Considerations, Algorithms and Architecture*, CWI, 2000-10

Jonas Karlsson, *Scalable Distributed Data Structures for Database Management*, CWI, 2000-11

2001

Silja Renooij, *Qualitative Approaches to Quantifying Probabilistic Networks*, UU, 2001-1

Koen Hindriks, *Agent Programming Languages: Programming with Mental Models*, UU, 2001-2

Maarten van Someren, *Learning as problem solving*, UvA, 2001-3

Evgueni Smirnov, *Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets*, UM, 2001-4

Jacco van Ossenbruggen, *Processing Structured Hypermedia: A Matter of Style*, VU, 2001-5

Martijn van Welie, *Task-based User Interface Design*, VU, 2001-6

Bastiaan Schonhage, *Diva: Architectural Perspectives on Information Visualization*, VU, 2001-7

Pascal van Eck, *A Compositional Semantic Structure for Multi-Agent Systems Dynamics*, VU, 2001-8

Pieter Jan 't Hoen, *Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes*, RUL, 2001-9

Maarten Sierhuis, *Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design*, UvA, 2001-10

Tom M. van Engers, *Knowledge Management: The Role of Mental Models in Business Systems Design*, VUA, 2001-11

2002

Nico Lassing, *Architecture-Level Modifiability Analysis*, VU, 2002-01

Roelof van Zwol, *Modelling and searching web-based document collections*, UT, 2002-02

Henk Ernst Blok, *Database Optimization Aspects for Information Retrieval*, UT, 2002-03

Juan Roberto Castelo Valdueza, *The Discrete Acyclic Digraph Markov Model in Data Mining*, UU, 2002-04

Radu Serban, *The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents*, VU, 2002-05

Laurens Mommers, *Applied legal epistemology; Building a knowledge-based ontology of the legal domain*, UL, 2002-06

Peter Boncz, *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*, CWI, 2002-07

Jaap Gordijn, *Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas*, VU, 2002-08

Willem-Jan van den Heuvel, *Integrating Modern Business Applications with Objectified Legacy Systems*, KUB, 2002-09

Brian Sheppard, *Towards Perfect Play of Scrabble*, UM, 2002-10

Wouter C.A. Wijngaards, *Agent Based Modelling of Dynamics: Biological and Organisational Applications*, VU, 2002-11

Albrecht Schmidt, *Processing XML in Database Systems*, UVA, 2002-12

Hongjing Wu, *A Reference Architecture for Adaptive Hypermedia Applications*, TUE, 2002-13

Wieke de Vries, *Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems*, UU, 2002-14

Rik Eshuis, *Semantics and Verification of UML Activity Diagrams for Workflow*

Modelling, UT, 2002-15

Pieter van Langen, *The Anatomy of Design: Foundations, Models and Applications*, VU, 2002-16

Stefan Manegold, *Understanding, Modeling, and Improving Main-Memory Database Performance*, UVA, 2002-17

2003

Heiner Stuckenschmidt, *Onotology-Based Information Sharing In Weakly Structured Environments*, VU, 2003-1

Jan Broersen, *Modal Action Logics for Reasoning About Reactive Systems*, VU, 2003-02

Martijn Schuemie, *Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy*, TUD, 2003-03

Milan Petkovic, *Content-Based Video Retrieval Supported by Database Technology*, UT, 2003-04

Jos Lehmann, *Causation in Artificial Intelligence and Law - A modelling approach*, UVA, 2003-05

Boris van Schooten, *Development and specification of virtual environments*, UT, 2003-06

Machiel Jansen, *Formal Explorations of Knowledge Intensive Tasks*, UvA, 2003-07

Yongping Ran, *Repair Based Scheduling*, UM, 2003-08

Rens Kortmann, *The resolution of visually guided behaviour*, UM, 2003-09

Andreas Lincke, *Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture*, UvT, 2003-10

Simon Keizer, *Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks*, UT, 2003-11

Roeland Ordelman, *Dutch speech recognition in multimedia information retrieval*, UT, 2003-12

Jeroen Donkers, *Nosce Hostem - Searching with Opponent Models*, UM, 2003-13

Stijn Hoppenbrouwers, *Freezing Language: Conceptualisation Processes across ICT-Supported Organisations*, KUN, 2003-14

Mathijs de Weerdt, *Plan Merging in Multi-Agent Systems*, TUD, 2003-15

Menzo Windhouwer, *Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses*, CWI, 2003-16

David Jansen, *Extensions of Statecharts with Probability, Time, and Stochastic Timing*, UT, 2003-17

Levente Kocsis, *Learning Search Decisions*, UM, 2003-18

2004

Virginia Dignum, *A Model for Organizational Interaction: Based on Agents, Founded in Logic*, UU, 2004-01

Lai Xu, *Monitoring Multi-party Contracts for E-business*, UvT, 2004-02

Perry Groot, *A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving*, VU, 2004-03

Chris van Aart, *Organizational Principles for Multi-Agent Architectures*, UVA, 2004-04

Viara Popova, *Knowledge discovery and monotonicity*, EUR, 2004-05

Bart-Jan Hommes, *The Evaluation of Business Process Modeling Techniques*, TUD, 2004-06

Elise Boltjes, *Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes*, UM, 2004-07

Joop Verbeek, *Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise*, UM, 2004-08

Martin Caminada, *For the Sake of the*

Argument; explorations into argument-based reasoning, VU, 2004-09

Suzanne Kabel, *Knowledge-rich indexing of learning-objects*, UVA, 2004-10

Michel Klein, *Change Management for Distributed Ontologies*, VU, 2004-11

The Duy Bui, *Creating emotions and facial expressions for embodied agents*, UT, 2004-12

Wojciech Jamroga, *Using Multiple Models of Reality: On Agents who Know how to Play*, UT, 2004-13

Paul Harrenstein, *Logic in Conflict. Logical Explorations in Strategic Equilibrium*, UU, 2004-14

Arno Knobbe, *Multi-Relational Data Mining*, UU, 2004-15

Federico Divina, *Hybrid Genetic Relational Search for Inductive Learning*, VU, 2004-16

Mark Winands, *Informed Search in Complex Games*, UM, 2004-17

Vania Bessa Machado, *Supporting the Construction of Qualitative Knowledge Models*, UvA, 2004-18

Thijs Westerveld, *Using generative probabilistic models for multimedia retrieval*, UT, 2004-19

Madelon Evers, *Learning from Design: facilitating multidisciplinary design teams*, Nyenrode, 2004-20

2005

Floor Verdenius, *Methodological Aspects of Designing Induction-Based Applications*, UVA, 2005-01

Erik van der Werf, *AI techniques for the game of Go*, UM, 2005-02

Franc Grootjen, *A Pragmatic Approach to the Conceptualisation of Language*, RUN, 2005-03

Nirvana Meratnia, *Towards Database Support for Moving Object data*, UT, 2005-04

Gabriel Infante-Lopez, *Two-Level Probabilistic Grammars for Natural Language Parsing*, UVA, 2005-05

Pieter Spronck, *Adaptive Game AI*, UM, 2005-06

Flavius Frasincar, *Hypermedia Presentation Generation for Semantic Web Information Systems*, TUE, 2005-07

Richard Vdovjak, *A Model-driven Approach for Building Distributed Ontology-based Web Applications*, TUE, 2005-08

Jeen Broekstra, *Storage, Querying and Inferencing for Semantic Web Languages*, VU, 2005-09

Anders Bouwer, *Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments*, UVA, 2005-10

Elth Ogston, *Agent Based Matchmaking and Clustering - A Decentralized Approach to Search*, VU, 2005-11

Csaba Boer, *Distributed Simulation in Industry*, EUR, 2005-12

Fred Hamburg, *Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen*, UL, 2005-13

Borys Omelayenko, *Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics*, VU, 2005-14

Tibor Bosse, *Analysis of the Dynamics of Cognitive Processes*, VU, 2005-15

Joris Graaumans, *Usability of XML Query Languages*, UU, 2005-16

Boris Shishkov, *Software Specification Based on Re-usable Business Components*, TUD, 2005-17

Danielle Sent, *Test-selection strategies for probabilistic networks*, UU, 2005-18

Michel van Dartel, *Situated Representation*, UM, 2005-19

Cristina Coteanu, *Cyber Consumer Law, State of the Art and Perspectives*, UL, 2005-20

Wijnand Derks, *Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics*, UT, 2005-21

2006

Samuil Angelov, *Foundations of B2B Electronic Contracting*, TUE, 2006-01

Cristina Chisalita, *Contextual issues in the design and use of information technology in organizations*, VU, 2006-02

Noor Christoph, *The role of metacognitive skills in learning to solve problems*, UVA, 2006-03

Marta Sabou, *Building Web Service Ontologies*, VU, 2006-04

Cees Pierik, *Validation Techniques for Object-Oriented Proof Outlines*, UU, 2006-05

Ziv Baida, *Software-aided Service Bundling - Intelligent Methods Tools for Graphical Service Modeling*, VU, 2006-06

Marko Smiljanic, *XML Schema Matching - Balancing Efficiency and Effectiveness by Eeans of Elustering*, UT, 2006-07

Eelco Herder, *Forward, Back and Home Again - Analyzing User Behavior on the Web*, UT, 2006-08

Mohamed Wahdan, *Automatic Formulation of the Auditor's Opinion*, UM, 2006-09

Ronny Siebes, *Semantic Routing in Peer-to-Peer Systems*, VU, 2006-10

Joeri van Ruth, *Flattening Queries over Nested Data Types*, UT, 2006-11

Bert Bongers, *Interactivation - Towards an e-cology of People, our Technological Environment, and the Arts*, VU, 2006-12

Henk-Jan Lebbink, *Dialogue and Decision Games for Information Exchanging Agents*, UU, 2006-13

Johan Hoorn, *Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change*, VU, 2006-14

Rainer Malik, *CONAN: Text Mining in the Biomedical Domain*, UU, 2006-15

Carsten Riggelsen, *Approximation Methods for Efficient Learning of Bayesian Networks*, UU, 2006-16

Stacey Nagata, *User Assistance for Multitasking with Interruptions on a Mobile Device*, UU, 2006-17

Valentin Zhizhkun, *Graph transformation for Natural Language Processing*, UVA, 2006-18

M. Birna van Riemsdijk, *Cognitive Agent Programming: A Semantic Approach*, UU, 2006-19