Agent Factory
A Rapid Development Framework for Agents

The ALPHA Programming Language
– Language Guide –

Robert J. Ross & Rem Collier

31 Aug 2004

Version: 0.2.1

www.agentfactory.com
The Agent Factory Working Group
e-mail: robertr at uni-bremen dot de
rem dot collier at ucd dot ie

Abstract

This document presents a description of ALPHA (A Language for Programming Hybrid Agents). ALPHA is a modern Agent Oriented Programming language with logical, deliberative and imperative features. This guide details ALPHA by treating each of its constructs in terms of syntax, usage and examples. However, this description is informal – a formal specification, once developed, will be provided elsewhere. In addition to detailing the low level language, the guide also introduces high level concepts including: roles, code re-use, and actuator, module, and perceptor constructs.

Document History

Version	Date	Author	Comments
0.1	09 Aug 04	robertr	Initial Version
0.2	29 Aug 04	robertr	Updated for Interpreter Implementation Details
0.2.1	31 Aug 04	robertr	Changed Document Title and Added Examples Chapter

Contents

1 Introduction 1

1.1 Agent Oriented Programming 1

1.1.1 Formal Agent Architectures 1

1.1.2 Agent-Oriented Programming Languages 2

1.1.3 Agent Architectures & Development Environments 3

1.1.4 Comparison to other Programming Paradigms 3

1.2 Formalising ALPHA 5

1.3 ALPHA, AF-APL & Language Variants 6

1.4 ALPHA and Agent Factory 6

1.5 Overview of Document 6

I ALPHA Language Description 8

2 An ALPHA Agent 9

2.1 Internal Structure 9

2.2 Execution Model for an ALPHA Agent 10

2.3 Levels of Control in ALPHA 12

3 Beliefs & Belief Rules 14

3.1 The Representation Format of ALPHA 14

3.2 Current Beliefs 15

3.3 Temporal Beliefs 15

3.4 The Belief Set 17

3.4.1 Belief Consistency 17

3.4.2 Intrinsic Beliefs 18

3.5 Initial Beliefs 18
3.6 Belief Set Querying & Belief Rules 18
3.6.1 Knowledge Inference – Belief Rules 19
3.6.2 Mental Functions 20

4 Perceptors **21**
4.1 The PERCEPTOR Construct 21
4.2 Perceptor Execution Model 22
4.2.1 Discussion 23

5 Actuators **25**
5.1 The ACTUATOR Construct 25
5.2 Actuator Execution 26
5.3 Uniqueness of Actuators 27
5.4 Altering the Mental State of the Agent 27
5.5 Behavioural Actuators 27

6 Plans **29**
6.1 The PLAN Construct 29
6.2 The Arrangement Constructs 30
6.2.1 SEQ – The Sequence Construct 30
6.2.2 PAR – The Parallel Construct 31
6.2.3 AND – The Random Order Construct 31
6.2.4 OR – The Non-Deterministic Choice Construct 31
6.2.5 XOR – The Deterministic Choice Construct 32
6.3 The Mental Constructs 32
6.3.1 The COMMIT Construct 32
6.3.2 The ADOPT_GOAL Construct 34
6.3.3 The ACHIEVE_GOAL Construct 34
6.3.4 The ADOPT_BELIEF Construct 34
6.4 The Imperative Constructs 34
6.4.1 TEST – The Belief Query Construct 35
6.4.2 FOREACH – The Universal Quantification Construct 35

6.4.3 The DO_BEHAVIOUR UNTIL Construct . . . 36
6.4.4 The TRY RECOVER Construct . . . 36
6.5 Plan Execution . . . 37

7 Commitments & Commitment Rules 39
7.1 Primary Commitment Adoption . . . 40
7.1.1 Initial Commitments . . . 40
7.1.2 Commitment Rules . . . 40
7.2 The Commitment Set . . . 42
7.2.1 Commitment Structures . . . 42
7.2.2 Ordering of the Commitment Set . . . 43
7.3 Commitment Structure Refinement . . . 43
7.3.1 Refining Commitments to Primitives . . . 44
7.3.2 Refining Commitments to Imperative Constructs . . . 49
7.3.3 Refining Commitments to Mental Constructs . . . 49
7.4 Reasoning about Commitments . . . 49
7.4.1 Beliefs about Succeeded and Failed Commitments . . . 52

8 Reactive Rules 53
8.1 Binding of Reactive Rules . . . 54
8.2 Firing of Reactive Rules . . . 54
8.3 Interference with Commitment Rules . . . 54
8.4 Reactive Rules & Reasoning . . . 55

9 Goals 56
9.1 Goal Adoption . . . 57
9.1.1 The ADOPT_GOAL Construct . . . 57
9.2 The Goal Set . . . 57
9.3 Goal Management . . . 57
9.3.1 Goal Fulfilment . . . 57
9.3.2 Goal Retraction . . . 61
9.3.3 Goal Management within the Agent Execution Cycle . . . 61
9.4 Reasoning about Goals . . . 61

10 Modules **62**

11 Some Other Language Features **63**

11.1 The ALPHA Namespace 63

11.2 Calling the Correct Actuator 63

11.3 Comments 64

II Using ALPHA **65**

12 Roles **66**

12.1 Rapid Prototyping with Roles 66

13 Compiling ALPHA Code for Interpretation **67**

13.1 The ALPHA Compilation Process 67

13.2 The ALPHA Auxiliary Files 67

13.2.1 Role Files 67

13.2.2 Plan Description Files 68

13.2.3 Actuator Description Files 69

13.3 ALPHA Roles & AF-APL 69

14 Example Code **70**

List of Abbreviations

ACL – Agent Communication Language

AF – Agent Factory

AF-APE – Agent Factory Agent Prototyping Environment

AF-APL – Agent Factory Agent Programming Language

AF-RE – Agent Factory Runtime Environment

ALPHA – A Language for Programming Hybrid Agents

(AO)P – (Agent Oriented) Programming

AOSE – Agent Oriented Software Engineering

CBSE – Component Based Software Engineering

DAI – Distributed Artificial Intelligence

DPS – Distributed Problem Solving

FIPA – Foundation for Intelligent Physical Agents

KQML – Knowledge Query and Manipulation Language

MAS – Multiagent Systems

(OO)P – (Object Oriented) Programming

Chapter 1

Introduction

Before describing ALPHA – A Language for Programming Hybrid Agents – this chapter establishes some background information on OOP, the ALPHA language philosophy, and the use of the language within the complete Agent Factory Framework.

1.1 Agent Oriented Programming

Much theoretical work has gone into the development of Agent Models which have proven to be useful in the development of complete AO programming languages and methodologies. In section 1.1.1 we will present a brief overview of these approaches. Section 1.1.2 then goes on to look at how Agent Oriented (AO) programming languages have developed over the past 15 years. The development of AO languages led to an interest in the provision of complete frameworks for agent system integration; A number of these frameworks are introduced in section 1.1.3.

1.1.1 Formal Agent Architectures

The analysis of agents and MAS can come from many different perspectives. From one point of view we can analyse the agent from the outside, ascribing qualities to describe the agent's interaction with other agents and its environment. Another approach, which originates in the *traditional* AI school is to formally model the internals of the agent. Here an agent is assumed to use a symbolic representation of the outside world, and that these symbolic representations are modelled as logical formulae, and that the agents operations are based on logical deduction and theorem proving based on these symbolic representations. A number of formal agent architectures have been developed in this way.

One important variant of these formal agent architectures are Belief Desire Intention architectures. These architectures have their basis in practical reasoning, namely the process of deciding moment by moment the actions which should be undertaken by the agent. Practical reasoning involves two separate processes: deliberation and means-end reasoning. Deliberation is the process by which the agent decides what goals it should

be attempting at a given time. Means-end reasoning then attempts to find a plan of action which can be used to achieve these goals. A BDI agent can essentially be broken down into three distinct parts. Firstly the agents beliefs represent the agent's knowledge (certain or otherwise) about itself and its environment. Desires can be thought of as the agent's goals, that is situations the agent has a long standing wish to bring about. Intentions are a subset of the agent's Desires and correspond to goals that the agent is actively trying to bring about. Like humans the agent can have potentially conflicting desires, but its intentions must be non-contradictory.

Many different formal BDI architectures have been developed which have been valuable in the subsequent development of agent-oriented programming languages. These approaches stemmed from work by Bratman [3, 2] on rational reasoning in humans. The best known early BDI architecture is the Procedural Reasoning System (PRS) of Georgeff and Lansky [12]. In the PRS, an agent is equipped with a library of plans which are used to perform means-end reasoning. Deliberation is achieved through the use of meta-level plans, which are able to modify an agent's internal structure at runtime, in order to change the focus of the agent's practical reasoning.

The initial work on PRS led Rao and Georgeff to further develop the logical framework [19, 11, 18]. Since then many other logical frameworks have developed based on these ideas. In [22] Wooldridge presents a BDI formulation based on the extended notion of commitment. In her study of communication and cooperation amongst a dynamic society of agents Haddadi extended Rao & Georgeff's initial models to a model of *Joint Commitment* [13]. This Joint Commitment model uses *pre-commitment* negotiations to allow agents to negotiate towards a joint commitment, before having to commit to the course of action. All these architectures were primarily developed from the standpoint of formal modelling, primarily dealing with the syntax and semantics of the underlying logic. In the treatments of these logics the construction of programming languages and interpreter systems served to demonstrate the completeness of the logics. A different approach described in the next section sees the programming language development as the main goal, with formalisation coming to serve the language and not vice versa.

1.1.2 Agent-Oriented Programming Languages

The logical frameworks described above directly led to the development of several agent implementations, e.g. PRS-Lite. These implementations allowed the design of an agent around a formal description. A related field saw the development of Agent-Oriented Programming Languages from the perspective of programming need. In [20] Shoham first advocated the agent oriented programming (AOP) paradigm, which viewed a complete system as being built out a number of different agents. AgentO was a complete programming language which could be used to describe an intentional agent[1] with a pseudo-BDI formulation. Agents are directly represented in terms of notions such as their beliefs, desires, and intentions. The purpose of such an analysis are two-fold. Firstly AO programs could directly be created with in-built reasoning and deliberative abilities, and secondly, that the construction of programs from the Intentional Stance,

[1]Such an interpretation is closely related to the analysis of a system from the *Intentional Stance* as advocated by Dennett [9, 10]. In should be noted that the Intentional Stance can be applied to all agents, regardless of their internal design. Thus, even a purely reactive agent can be described with intentional attributes.

rather than the more traditional Design or Physical stances, allows the analysis of complex systems with high level pseudo psychological qualities. In AgentO, an agent is specified with set of capabilites, a set of initial beliefs, a set of initial commitments, and a set of commitment rules. The use of commitment rules are the key deliberation process available to an AgentO agent, and help to define the interface between what the agent believes and what the agent wishes to do. A recent AO language which has attempted to built on the success of Agent0 is 3APL [7]. This language has many of the basic features of Agent0 but also has a well defined operational semantics provided by transition states. Other languages worthy of mention include Rao's AGENTSPEAK [17], and PLACA by Thomas [21]. ALPHA, described here, is closely modelled around Agent0 but with a logical framework similar to that developed by Wooldridge in [22].

1.1.3 Agent Architectures & Development Environments

The development of AO languages and formulation of agent models for real world programming, invariable led to an interest in complete frameworks which could be developed both to provide development time and run-time support for agents. Unfortunately there are many different frameworks, and each of them support different views of what type of agent they are dealing with. For example JADE [1] provides foundations for multiple reactive agents, while ZEUS [16] and OAA [15] provides a general framework for the integration of legacy systems. Toolkits and frameworks have also been developed around true AO languages. JACK [4] supports the fabrication of agents through a combination of capabilites, events, belief sets, and plans. Specifically, JACK uses a set of custom extensions to Java to create agent designs. These designs, are then compiled into pure Java code that can be executed directly. Although a complete framework, JACK's AO language JAL is limited in its AO abilities.

An agent framework which is similar to JACK is Agent Factory [5]. Developed in University College Dublin, Agent Factory is a complete agent prototyping and development environment. Agents are usually designed around the Agent Factory Agent Programming Language (AF-APL). The language is very similar in nature to Shoham's AgentO, but contains a formal treatment of commitment similar to Wooldridge's. An AF-APL agent is defined in terms of its beliefs, commitments, belief inference rules, commitment rules, actuators, plans and perceptors. These constructs correspond closely to the *common sense* understanding of their meanings, but special attention should be given to the concept of a commitment, and its implications. A commitment is essentially a promise made by an agent to itself or other, to attempt to perform an action. These actions can be simple, matching one actuator call, or can be complex involving a plan like sequencing of lower actions. An AF-APL agent is modelled in terms of its commitments, and a formal analysis of the management of these commitments.

1.1.4 Comparison to other Programming Paradigms

A commonly posed question is whether there are really any differences between Component Based and Agent Oriented Based Software Engineering. A similar question is the more elementary 'What is the difference between an object and an agent'. These questions relate to a common misconception that any program or software component should be labelled as a software agent for the sake of simplifying complex system de-

scriptions. Clear distinctions can be drawn between the agent and object or component, and these distinctions are important to identify before the advantages to agent oriented design can be appreciated.

Agent-Oriented Versus Object-Oriented Design

In [23] Wooldridge presents a clear discussion of the differences between agents and objects. From this discussion we can conclude that agents:

- **Request Action rather than Command.** Due to their autonomous nature agents have a choice as to whether they should perform any request made of them. An agent only responds to a request when it is in its best interest, whereas an objects method will always be called.
- **Have a Dedicated Thread of Execution.** Given their pro-active (or non-passive) nature, agents are typically implemented with their own thread of execution and control loop.
- **Engineered Autonomy.** By definition agents, are designed to have autonomous characteristics.

It might be argued that any object can be engineered to show all the characteristics of an agent. This is absolutely true, and unsurprisingly the object will then by definition be an agent. In fact it should be noted that many agent development languages simply model an agent as a very complex object which has had mechanisms for autonomy, pro-activity and situatedness built in. Recent multimedia and web development has given rise to interest in so called *Active Objects* . These *active objects*, are a variant on components, which are covered in the next section.

Agent-Oriented Versus Component Based Design

Distributed computing has led to a greater interest in the topic of Component Based Software Engineering (CBSE). CBSE is in many cases similar to Object Based Software Engineering. However Components are larger, or more coarse grained, than objects. A component will often be made out of a number of different objects or even programs. Related to CBSE is the industrial term middleware, which is a general term for programming that serves to "glue together" two separate and often already existing programs, i.e. a layer used in the integration of legacy systems. A common application of middleware is to allow programs written for access to a particular database to access other databases.

Components have many of the characteristics of intelligent agents, namely usefulness in legacy component integration, course grained decomposition, message based communication. These components are however only course grained objects and can still be distinguished from agents by:

- **Not Pro-Active** – Like objects, components are often static in nature. They simply sit waiting for instructions or actions to be performed on them.

- **No Engineered Autonomy** – The component is only a container for a legacy system and adds no value to the system itself. Intelligent Agents have their own inherent intelligence which improves the overall usefulness of the agent as a whole.

1.2 Formalising ALPHA

ALPHA is directly derived from the AF-APL programming language as described in [5]. ALPHA was forged out of a number of changes that were made to both the core AF-APL language and the Agent Factory interpreter over a period of time. A formal specification of the language can provide a number of clear advantages:

- **Allows Multiple Platform Usage** – A detailed description of the behaviour and syntax of ALPHA is very important to allow us to use the language on a number of different platforms and interpreters.
- **Allows Comparisons & Evaluation** – With a clear understanding of the language, we can compare and contrast it with other AO languages.
- **Makes the AO Features of Agent Factory Clear** – ALPHA and AF-APL's features are essential to distinguishing the Agent Factory Framework from a Middleware infrastructure.
- **See the Whole Picture** – By going to the effort of laying out a formal description, we gain a more complete picture of ALPHA – and hence, identify features which might be missing.
- **True Formalism** – Informal specifications are inherently weak and can easily fail to describe the logic completely – by using a formal specification we can guarantee that all aspects of the language are specified.
- **Formal Model Checking** – We can analyse the formal model to search for errors in the basic design in the language. This allows us to make an AO language which is powerful, yet not plagued by inconsistency.
- **Usefulness in Application** – An AO language or development environment which is well specified, is immensely more trustable in a real world environment than an unspecified environment where much is assumed.

This document is the first step in providing a formal description of ALPHA. This informal description is intended to act as a catalyst for discussion, enabling the development of an agreed language description, which can then be formally described.

AF-APL was originally specified with a formal logic: λ_{af}. It is, as yet, unclear how ALPHA will be formalised. In any case, the development of the formal treatment will act as a tool for the ALPHA language, rather than an academic exercise in itself.

1.3 ALPHA, AF-APL & Language Variants

We do not believe that any one current Agent Oriented Programming language is capable of delivering agents which meet all possible needs. In particular, we acknowledge differences on: (a) reactivity versus deliberation; and (b) resource limitations in agent environments. The language ALPHA is a derivative of AF-APL, which can be broadly described as a faster, but less featured agent oriented language. ALPHA can be viewed as an experimental language that aims to encapsulate many of the stronger notions of BDI agents.

Other variants on the language may be considered in future, these variants include:

- **Static ALPHA** – A variant which does not allow the run-time adjustment of mental objects apart from beliefs i.e. the adjustment of commitment rules, actuators or perceptors during run-time would not be allowed. This would result in a less dynamic but more predicatable agent design.
- **Threadless ALPHA** – A variant of the language which would allow agents to exist on platforms where there are bounds on thread resources.

ALPHA is directly derived from, and shares much in common with Collier's AF-APL language [5]. ALPHA inherits AF-APL's basic notions of belief, commitment and action, but adds explicit goals, an enhanced actuator and perceptor model, reactive rules, an asynchronous execution model, and a number of plan operators.

ALPHA is partially backward compatible with AF-APL. Specifically, an AF-APL agent type is valid as an ALPHA role, and may be used to compile an ALPHA agent. However AF-APL code cannot be used directly by the ALPHA interpreter.

1.4 ALPHA and Agent Factory

The Agent Factory Framework [5] is a complete agent prototyping environment, providing development facilities similar to those of OAA [15] and JACk [4]. Unlike these environments Agent Factory supports agents based based on strong intentional agent oriented programming language similar to Shoham's AgentO [20], Goal Directed 3APL [8], and Wooldridge's formulations for intentional rational agents [22].

In addition to supporting an ALPHA interpreter plugin, Agent Factory provides a large suite of tools for the development and deployment of ALPHA agents, including: prototype agent designs; language interpreters and compilers; a fully functional IDE; and agent and platform viewers. (See Figure 1.1). These tools are beyond the scope of this language guide, but have previously been introduced in [6].

1.5 Overview of Document

The document is structured in two parts to allow clear differentiation between the core language, and details of how the language may be used. In Part 1, chapter 2 introduces

Figure 1.1: The Agent Factory Framework

ALPHA at a high level. This description of the agent as a whole is followed by chapters 3 through 11 with detailed descriptions of ALPHA's features. Part II looks at how ALPHA is used to make real agent application. Chapter 12 looks at how roles are used with ALPHA, before Chapter 13 shows how Roles, Actuators and Perceptors fit into the agent compilation process.

Part I

ALPHA Language Description

Chapter 2

An ALPHA Agent

This chapter provides a high level description of an ALPHA agent; looking at the core components of an ALPHA agent, the agent's execution model, and the levels of control available to these agents.

2.1 Internal Structure

ALPHA is an AO programming language, and as such explicitly encodes many features necessary for the design of a complete autonomous agent. Figure 2.1 depicts the key features of an ALPHA agent. The agent can be broadly split into three sets of components: the mental state, the actuators, and the perceptors. An actuator is a pieces of external code (typically Java or C) that allow the agent to manipulate its environment. Similarly, perceptors are pieces of external code that update the agent's beliefs about the state of its environment. These actuators and perceptors are represented within ALPHA code as external code; this representation is similar to the representation of native functions used in Java code. The agent's mental state components are the high level notions used to program and manipulate the agent. ALPHA's mental state elements include:

- **Beliefs** – The knowledge held by the agent about its environment.
- **Belief Rules** – Rules which allow the agent to infer new beliefs from other beliefs.
- **Reactive Rules** – Rules which allow the agent to invoke actions immediately based on current beliefs.
- **Plans** – Recipes for action which usually make use of actuators.
- **Commitments** – Promises made by the agent to perform certain actions.
- **Commitment Rules** – Rules which cause an agent to adopt a commitment when a certain belief state has occurred.
- **Goals** – States of the world that the agent wishes to bring about.

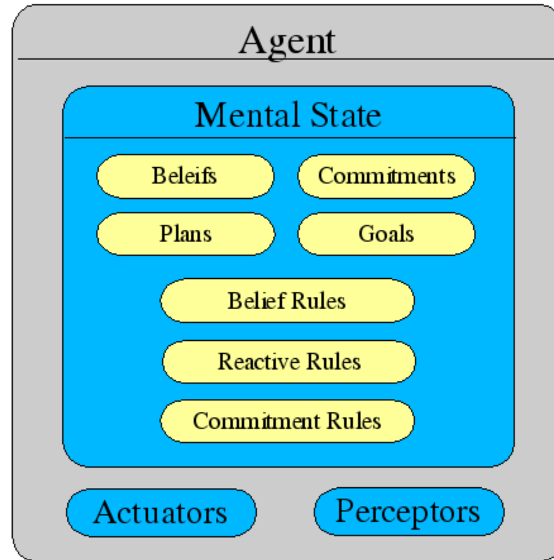


Figure 2.1: The Salient Features of an ALPHA Agent.

2.2 Execution Model for an ALPHA Agent

With plans, ALPHA has some superficial features of an imperative programming language, i.e., an ALPHA agent can execute a structured sequence of actions. However, ALPHA is very different from an imperative programming language. At a high-level, ALPHA has more in common with a declarative or logical programming language, and it is only at the lower level that the imperative nature of plans can be seen. The ALPHA execution model, presented below, is based on the AF-APL execution model developed in [5]. The original model has been modified to provide an asynchronous execution model that provides the agent with reactive and deliberative capabilites.

An ALPHA agent's Life Cycle is built out a of number of Execution Cycles as shown in figure 2.2. Rather than forcing users to define their own execution or deliberation cycle, ALPHA provides a well defined execution model which reflects the basic needs of a hybrid agent. Unlike old fashioned, Sense – Plan – Act approaches, the model is asynchronous, guaranteeing the continued operation of an agent – regardless of potential locking of 3rd party actuator and perceptor code. During an execution cycle perceptors are first fired, generating new beliefs for the agent's belief set. Reactive rules are then triggered to implemented basic reflective behaviours. The Commitment Management process then follows; firing commitment rules to generate new primary commitments, and refining commitment structures through the application and partial execution of plans and actuators. Figure 2.3 presents a pseudo code description of an agent's Execution Cycle.

See [5] for motivation for the basic form of this Execution Cycle. Notable differences between this Execution Cycle and the AF-APL formulation include:

- **Deliberation** – Goals which drive a means end reasoning process are now an explicit construct within the language.

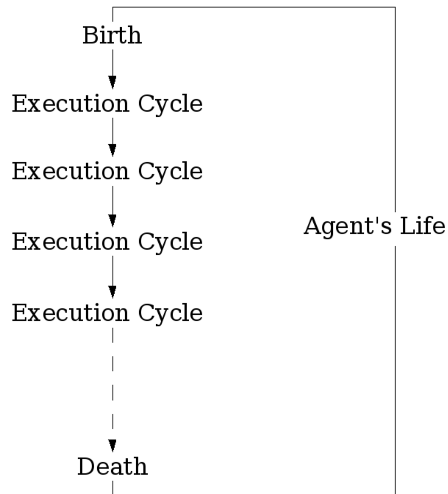


Figure 2.2: The Life Cycle of an ALPHA Agent

```
Belief Management
   Perceptor Management
       Empty Perceptor Queues into Belief Set
       Fire Each Perceptor in its own Thread
    Temporal Belief Management

Reactive Rule Management
  Fire resultant actions in their own threads

Deliberation / Goal Management
  Drop Any Goals which have been met
     Drop any subsequent Commitments
  Apply means end analysis rules once to each goal.

Commitment Management
  Generate New Commitment Set
  Take Each Commitment in turn
       Manage Commitment
         Fire Actions (can result in the acquisition of A Priori Beliefs)
```

Figure 2.3: Pseudo Code for the Execution Cycle of an ALPHA Agent

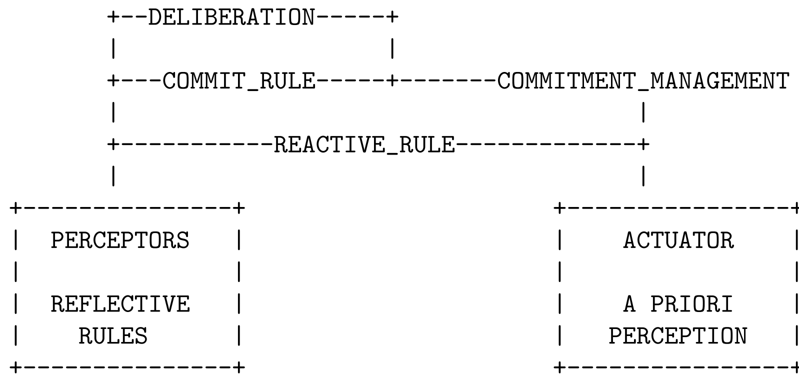


Figure 2.4: Levels of Control within an ALPHA Agent

- **Threading** – Perceptors, actuators and means-end reasoning are all performed in independent threads. This gives the resultant ALPHA greater flexibility and robustness.
- **Commitment Management** – Although not shown here, the Commitment Management process used in ALPHA is substantially different from that defined for AF-APL. The commitment management process is discussed further in 7.

2.3 Levels of Control in ALPHA

Through constructs that provide different levels of control, ALPHA is a powerful programming language which can implement a range of agents with deliberative and reactive abilities. Rather then using either a `SENSE-PLAN-ACT` or purely reactive design, ALPHA provides an agent which has a range of control levels. These are shown diagrammatically in figure 2.4. Moving from the most reactive to the most deliberative, the control levels are:

- **Reflective Control** – Perceptors can directly code reflective responses to external stimuli. These reflexes are similar in nature to a reflex action in humans. There is no symbolic representation for such a reflex action in ALPHA, and as such is not an explicit part of the ALPHA language. The design of the reflective code might however *notify* the higher level agent code through the adoption of belies about the reflexive action which was made.
- **Reactive Control** – Reactive Controls are provided in ALPHA by Reactive Rules. Unlike reflexive controls, reactive controls are implemented at the level of symbolic representation within ALPHA and can hence be considered of a higher grain intelligence where the reaction is dependant in some way from possibly multiple actuators or belief inference. Reactive Rules are however less intelligent then Commitment Rules and have no facilities for error recovery or complex plan execution. Reactive Rules are acted on by the agent before either Deliberation or Commitment Management.

- **Commitment Rules & Commitment Management** – The management of commitments which are generated by Commitment Rules provide a more intelligent level of agent control. Commitment rules relate a set of beliefs to a commitment to an action or plan which must be attempted. The agent attempts to achieve such a commitment through the Commitment Management process. Unlike a simple plan execution approach Commitment Management provides in-built facilities for error recovery and is conceptually linked with high level deliberation.

- **Deliberation** – Deliberation is the highest level of control available to an ALPHA agent. Deliberation involves the use of explicit Goals to motivate the agents actions. Means-end reasoning is used to relate a Goal to plans which the agent can then *commit* to.

Chapter 3

Beliefs & Belief Rules

This chapter introduces ALPHA's basic notions of Beliefs, detailing: ALPHA's knowledge representation language; current and temporal beliefs; belief rules; and the agent's belief set.

3.1 The Representation Format of ALPHA

An agent's beliefs denote what the agent believes to be true about itself or its environment. Before detailing beliefs in detail, we introduce the knowledge representation language that underpins all logical constructs in the ALPHA language.

Underlying all of ALPHA's logical language constructs is a common language or representation format (RF). This format is used in two principle contexts: (a) to represent predicate formulae for beliefs and goals, and (b) to represent action, perception and plan interfaces. This format must support the representation of known objects, unknown objects, and composite objects. This is achieved through three components:

- **Constants -** Constants are labels used within the logic to represent objects of interest in the agent's environment. Within ALPHA, any aggregation of ASCII characters may be a constant so long as the first ASCII character is not a question mark. Also when a constant contains whitespace, braces, double quotes or commas, that constant must be enclosed in double quotes. Further if the constant itself contains a double quote it must be preceeded by the escape character \. Examples of allowable constants include: X, 22, someConstant, hello_world, "i have data = \"my data\" ".
- **Variables -** Variables are used to identify unknowns. They are used in queries of the agent's beliefs, action and plan interfaces. Within ALPHA, any aggregation of alpha-numeric, underscore, and hyphen characters may be a variable so long as the first character is a question mark, and that all other characters are not numerals. White space, commas and braces are explicitly not permitted in variable. Examples of allowable variables include: ?Agent, ?x, ?Y, and ?user.
- **Composites -** Composites define aggregations of constants and variables to represent actuators, perceptors or to describe relationships between objects. They

are represented in the usual form as a functor combined with a set of arguments (constants or variables). Allowable functors in ALPHA are alpha-numeric strings containing at least one letter, no commas, whitespace, braces or double quotes, where the first character is not a question mark '?', and where the functor is not completely composed of numerals. For example, happy(?x) is a composite that defines the unary relationship that ?x is happy. Conversely, composites such as eat(?x) can be used to identify the action of eating ?x.

Within ALPHA, valid formulae are either constants or composites and are known as **RF atoms**. Variables may only be used within composites. Two particular constants used within the system are TRUE and FALSE. These evaluate to true and false respectively. Apart from true and false, no formulae of the representation format should appear outside the scope of the constructs identified in chapters 3 to 11

3.2 Current Beliefs

An agent's beliefs represent what the agent believes to be true about itself or its environment, and are identified within ALPHA with the BELIEF construct. The construct takes one argument, an RF atom, to represent what the belief is about. For example, the belief that Rem is happy may be represented with the statement:

```
BELIEF(happy(Rem))
```

Some other examples of beliefs include:

- BELIEF(partner(rem)) – rem is a partner.
- BELIEF(likes(rem, beer)) – rem likes beer.
- BELIEF(told(bob, jump)) – bob has been told to jump
- BELIEF(bid(fred, 50)) – fred has bid 50
- BELIEF(informed(terry, hello)) – terry has informed the agent of hello
- BELIEF(heard_message(rem, "where are you")) – rem has received a message

The belief construct is used widely within ALPHA. Its primary use is to define the current beliefs of the agent. These are all the beliefs that are true at a particular point in time. In addition to this, the belief construct is used in conjunction with a number of logical connectives and temporal operators as shown in the following sections.

3.3 Temporal Beliefs

By default, an agent holds a belief for a particular point in time, but the following language constructs are provided to allow a belief to persist over time.

The ALWAYS Operator

This construct specifies a belief that must be adopted immediately, and which should persist until explicitly removed from the agent's belief set. The operator takes one argument, the belief to be adopted at the current time point. To represent the belief that Rem always likes beer, it is possible to use the ALWAYS construct:

```
ALWAYS(BELIEF(likes(Rem, beer)))
```

Following adoption, the agent's belief set includes both the original *ALWAYS Belief* and an instance of the appropriate current belief. All beliefs – including temporal beliefs – can be removed from the agent's belief set; Explicit removal of an ALWAYS belief will result in both the ALWAYS belief, and the appropriate current belief being removed immediately.

The NEXT Operator

This construct specifies a belief that must be held by the agent during the next agent execution cycle. However, the construct itself does not persist (i.e. it exists only for one iteration of the agent interpreter). The construct has one argument, the belief to be adopted at the next time point. The belief that Rem will be a partner of the agent during the next execution cycle is represented as:

```
NEXT(BELIEF(partner(Rem)))
```

The UNTIL Operator

This construct specifies a belief that must be adopted during the current execution cycle, and which must persist until either a defined belief becomes true, or until it is explicitly removed from the agent's mental state. The construct has two arguments: the first argument is the belief to be adopted; while the second is the condition under which the commitment should be dropped from the agent's belief set. To represent Rem's belief that Terry is a partner until Rem believes that Terry is not a student any more, it is possible to use the UNTIL construct:

```
UNTIL(!BELIEF(student(Terry)), BELIEF(partner(Terry)))
```

UNTIL beliefs are a useful way of providing belief consistency. An agent can believe something, as long as it does not believe something that contradicts the original belief. For example, the agent may decide to believe that the sky is blue until it knows otherwise; this would be represented as[1]:

```
UNTIL(BELIEF(colour(sky,blue)), BELIEF(colour(sky,?some_colour)))
```

[1] What I want to say is that the belief should be dropped when I believe that the sky is some other colour, but I don't know if the present design will take care of it. I will need an equality operator or equivalent

The nesting of temporal constructs is also permitted. For example, the following temporal belief states that at the next time point, the agent will always believe that Rem is its partner:

```
NEXT(ALWAYS(BELIEF(partner(Rem))))
```

Thus, the current belief `BELIEF(partner(Rem))` will be adopted two execution cycles after this temporal belief is adopted.

3.4 The Belief Set

To this point, we have inferred that an agent's mental state contains a number of beliefs at any one point in time. We refer to this collection of beliefs as the agent's belief set, and it is a conjunction of all current and temporal beliefs held by the agent.

There are a number of limitations on what an agent can believe, and hence on the form of beliefs that can be adopted and inferred. These restrictions, defined to reduce computational complexity, include:

- **No Negative Beliefs** – Neither negative beliefs nor beliefs about negative RF atoms are allowed. Thus, neither `!BELIEF(partern(Jonny))` nor `BELIEF(!partner(Jonny))` may be added to the agent's belief set – however, the former construction is a valid construction for querying the belief set (See Below).
- **No Nested Beliefs** – An agent cannot have beliefs about its own beliefs or other agent's beliefs. This first order logic restriction places a limitation on the agent's reasoning skills, but is a computational necessity.
- **No Unbound Beliefs** – Since adopted beliefs must be about positive RF atoms, Beliefs about RF atoms containing free variables cannot be contained within the Agent's Belief Set.

3.4.1 Belief Consistency

Some agent-oriented, and knowledge representation languages have allowed explicit belief consistency checking. The purpose of such checks is to put constraints on what beliefs can be adopted by the agent at the given point in time. ALPHA does not include such this form of consistency check since they can potentially lead to a situation where an agent could not adopt valid beliefs. Instead, ALPHA provides belief consistency checking through the use of belief inference rules (see section 3.6.1) and the UNTIL operator (see above). Also, since negative beliefs are not allowed in the belief set, directly contradictory beliefs are not a problem; that is, an agent cannot believe at the same time that it likes jam and it does not like jam.

3.4.2 Intrinsic Beliefs

An ALPHA agent has a number of intrinsic beliefs that allow the agent to reason about its own goals, commitments and elementary issues such as self, time, truth and arithmetic.

Beliefs About Truth

Since RF includes the constants *true* and *false* , the agent's belief set includes `BELIEF(TRUE)`. This belief is commonly checked against in default commitment maintenence conditions and plan pre-conditions. Similarly, since *false* is defined in RF, inquiring as to whether the agent believes FALSE will always return false.

Since *true* and *false* are defined as constants in RF, adding or removing BELIEF(TRUE) and BELIEF(FALSE) are meaningless. Adding BELIEF(TRUE) will always result in true, while adding BELIEF(FALSE) will return false. Conversely, removing BELIEF(TRUE) will return false, while removing BELIEF(FALSE) will always return true.

3.5 Initial Beliefs

The initial beliefs of the agent can be explicitly coded in the ALPHA design. Since the agent's Belief Set can include any positive current belief or beliefs used with the Temporal Operators, allowable initial beliefs include:

```
BELIEF(happy(Rem));

NEXT(ALWAYS(BELIEF(partner(Rem))));

UNTIL(!BELIEF(waltzing),BELIEF(partner(Rem)));
```

3.6 Belief Set Querying & Belief Rules

Some ALPHA constructs make explicit queries of the Belief Set to test whether certain constraints hold; For example, queries are made once per execution cycle to decide whether commitment rules should be triggered. Programmatic queries of the Belief Set are also possible, using either the TEST planning operator, or directly in actuator or perceptor code.

A belief set query is a conjunctions of positive or negative current beliefs containing free or bound variables. The most simple belief set query takes the form:

```
BELIEF(eating(apple))
```

where the query will return true if, and only if, the belief set entails `BELIEF(eating(apple))`. Alternatively, the following will return true if the agent believes that it has any friends.

```
BELIEF(friend(?name))
```

Queries can also be used to check the negation of beliefs. For example, the following will return true if, and only if, the agent does not belief that it is eating an apple:

```
!BELIEF(eating(apple))
```

It should be noted that although the belief set cannot contain negative beliefs, a belief set query can be negative.

As well as being able to determine the truth value of a predicate, the agent's belief set can also be queried for all possible beliefs that match a query. This can be done either in an ALPHA plan construct, or programatically through the actuator interface. Thus, if the agent's belief set entails a given query, then a set of all matching beliefs is returned. If the query is not entailed, the query is interpreted as false.

3.6.1 Knowledge Inference – Belief Rules

As well as containing simple predicates, the agent's knowledge representation also includes a number of belief inference rules, or simply Belief Rules. For example:

```
BELIEF(A) & BELIEF(B) => BELIEF(C);
```

states that if the agent believes both A and B, then, by inference, it also believes C.

The Belief Rule has the general form:

```
BASIS_SENTENCE => INFERRED_BELIEF;
```

where BASIS_SENTENCE is a belief query as introduced above, and INFERRED_BELIEF is a logical consequence of that belief. INFERRED_BELIEF is entailed by the agent's belief set if, and only if, BASIS_SENTENCE is entailed by the agent's mental state. Both the BASIS_SENTENCE and the INFERRED_BELIEF may contain free variables. However, any free variables in the INFERRED_BELIEF must also be free variables in the BASIS_SENTENCE. Therefore, the following is not a valid belief rule, since '?venue' cannot be bound to the BASIS_SENTENCE:

```
BELIEF(friend(?x)) & BELIEF(having_party(tomorrow))
   => BELIEF(should_invite(?x,?venue);
```

As well as querying the agent's belief set, the BASIS_SENTENCE can also contain terms that query the agent's commitment and goal sets. See sections 7 and section 9 for examples of this usage.

3.6.2 Mental Functions

In addition to queries of propositional relationships stored in the belief set, ALPHA defines the following three mental functions that allow the agent to make basic equality tests:

- **Equivalence** – equal(?x,10)
- **Greater Than** – greaterThan(?x,10)
- **Less Than** – lessThan(?x,10)

These functions can only be used in a restrictive from in belief query sentences. Free variables in the functions (?x in this case) must have been bound by something in the preceeding belief query statement; otherwise, the relational operator evaluates to false.

These three are the only relational operators defined within ALPHA. New operators cannot be designed within ALPHA, since badly designed operators could seriously impact on the belief query model.

For example, an agent can adopt beliefs that it can drink alcohol with any of its friends, who are above the age of 18, with the following belief rule:

```
BELIEF(friend_age(?friend,?years)) & BELIEF(greaterThan(?years,18))
   => BELIEF(canDrinkAlcohol(?friend));
```

When applied to inappropriate terms, these operators will always return false. For example, lessThan(Rem,10) evaluates to false since 'Rem' is not a numeral.

Chapter 4

Perceptors

An agent's perceptor is a piece of code that generates beliefs about the agent and its environment. This chapter introduces the perceptor construct and the perceptor execution model.

4.1 The PERCEPTOR Construct

Within ALPHA the PERCEPTOR construct is used to declare each of the agent's perceptors. The argument takes the form:

```
BEGIN{PERCEPTOR}
IDENTIFIER ie.ucd.myProject.myPerceptor;
CODE ie.ucd.myproject.perceptors.Perceptor.class;
END{PERCEPTOR}
```

where the two definition sections are:

- **IDENTIFIER** – The IDENTIFIER of the perceptor is a unique identifier within the ALPHA namespace.
- **CODE** – CODE defines a piece of *external code* that can be used to perform the perception task. Multiple CODE declarations can be defined, but only one piece of code is used on any platform. Perceptor's code adds new beliefs to the agent's belief set through defined pipes into the belief set. Perceptors in Java and C are supported directly by ALPHA, while many other languages and applications can be supported if they have an exportable C interface.

As indicated, for any given perceptor, any number of CODE sections can be defined; this can be useful when dealing with multiple hardware platforms. Take for example a mobile agent that crosses between a high performance PC, and a computationally limited handheld device. Conceivably, when the agent moves to the low-spec platform, it may need to switch to a more basic form of perceptor. For this agent, we can define a perceptor with two CODE implementations as follows:

```
BEGIN{PERCEPTOR}
IDENTIFIER ie.ucd.mobileAgent.environmentPerceptor;
CODE ie.ucd.mobileAgent.highPowerPerceptor.class;
CODE ie.ucd.mobileAgent.lowPowerPerceptor.o;
END{PERCEPTOR}
```

When the perceptors are loaded at the agent's birth, perceptor CODE sections are loaded into memory in sequence. If the first perceptor cannot be instantiated, an attempt is made to instantiate the second perceptor, and so forth. If no perceptor CODE blocks can be loaded, the agent will run without the specified perceptor being initialised.

Once a perceptor is included in an agent design, there is no need to reference the perceptor again; it will be called once per execution cycle of the agent throughout the agent's life cycle.

In ALPHA external perceptor code: (a) takes no arguments; and (b) has no return value. As mentioned above, both Java and C external perceptors are supported.

Perceptors can add and remove beliefs from the agent's belief set. In Java this is done with the aid of a number of methods inherited from a basic Perceptor class, while in C a library of functions is provided to achieve the same goal. Perceptors are not otherwise allowed to directly alter the agent's mental state since all other manipulation should be rationally decided based on the agent's beliefs.

4.2 Perceptor Execution Model

ALPHA's perceptor execution model is an asynchronous model based loosely on that defined for AF-APL. The main advantage to this asynchronous model is that the agent's execution cycle never becomes locked due to third party code. AF-APL's execution model called perceptors directly within the Agent's main execution thread. The advantage to such a model was that perceptor execution has a clear stepwise model matching the agent's reasoning cycle. On the other hand, this puts a clear constraint on the robustness of the complete agent. If a perceptor were to contain third party code which blocked for some reason, then the agent's execution cycle would become stuck at this perception execution step.

We now define a perceptor as an external piece of code which, unlike actuators, is constantly running or repeatedly invoked to update the agent's view of the world. During each of the agent's execution cycles, an attempt is made to trigger the perceptor implementation. If the perceptor is not already running, the perceptor is fired off in its own thread. It the perceptor is already running, then no attempt is made to re-invoke the perceptor, or to terminate the current still running invocation.

Rather than adding beliefs directly to the agent's mental state, each perceptor adds new beliefs to a unique queue which is defined alongside the perceptor interface. This belief queue is emptied by the agent control loop once per execution cycle, just prior to attempting to fire the perceptor again. The contents of this belief queue are then added to the agent's Belief Set. The queue is emptied at that time. In pseudo-code the revised execution model looks like;

```
// Perception
FOREACH(PERCEPTOR(?X)) {
   Add X.queue to Agent.BeliefSet;
   if X.isRunning {
      // do nothing
   }
   else {
      start X.method in seperate thread;
   }
}
// Reasnoing
...
// Commitment Management
...
```

4.2.1 Discussion

ALPHA's perceptor model is quite distinct from that previously seen. Indeed, two perceivable disadvantages to this revised model can be identified:

- **Synchronisation** – The major drawback to this design is that we loose a certain amount of synchronisation between perceptors. Take the example of a belief rule such as

  ```
  BELIEF(A) & BELIEF(B) => BELIEF(C);
  ```

  Now suppose that Perceptor-A was typically responsible in BELIEF(A) being adopted when A is true in the real world; And Perceptor-B was typically responsible for adoption of BELIEF(B) when B is true in the real world. AF-APL synchronised perceptor model might allow for a situation where the agent would believe both BELIEF(A) and BELIEF(B) in the same execution cycle (as long as A and B were true in the real world), and hence infer BELIEF(C) to be true. Within ALPHA's model, if Perceptor-A actually took two execution cycles to detect and report A, while Perceptor-B took less than one execution cycle, then the agent could not believe both BELIEF(A) and BELIEF(B) at the same time, and henae could not make the inference BELIEF(C).

- **Reactivity** – Another criticism of ALPHA's perceptor model is that the revised design makes the agent slightly less reactive in design. Rather than a perceptor being executed immediately resulting in the agent reasoning on the beliefs immediately, the result of perception at time n will only be evaluated at time n+1.

The first criticism of the new perception model can be easily critiqued. In the real world, we rarely design agents to have behaviours which are dependant on synchronised perceptors (or indeed synchronised actuators). One of the cornerstones of agent design is that we can not make such guarantees about an agent, and that perceptors are noisy and often lead to late or imperfect data. Further, the use of ALWAYS, NEXT and

UNTIL constructs already argument reduce this problem since they allow controlled persistence in the agent's beliefs, thus allowing the agent to have a more *fuzzy :-)* view of the world.

The second argument, regarding the reduced reactivity of the agent, is also easily dismissed. Since the perceptor execution is out of the main control loop, perceptor execution time is now far faster, hence meaning that the real time difference between dealing with information in execution cycle n+1 instead of execution cycle n is negligible. The advantage that is gained by guaranteeing that the agent control cycle can never block, this reactivity limitation is completely outweighed.

Chapter 5

Actuators

Actuators are pieces of external code that the agent uses to manipulate its environment. This chapter introduces actuators, detailing the ACTUATOR construct, and discussing a number of actuator execution issues.

5.1 The ACTUATOR Construct

The actuator construct is used to represent the actuators of the agent within the agent program. The ACTUATOR construct takes the following form:

```
BEGIN{ACTUATOR}
IDENTIFIER ie.ucd.myProject.myActuator(?aguement1,?arguement2);
PRE BELIEF(TRUE);
POST BELIEF(i_did(x)) | BELIEF(i_did(y);
CODE ie.ucd.myproject.actuators.Actuator.class;
END{ACTUATOR}
```

An Actuator construct consists of the following sections:

- **IDENTIFIER** – The IDENTIFIER uniquely defines the actuator in the ALPHA namespace. The identifier explicitly defines the arity of the actuator (the number of variables taken by the actuator). Variables supplied must be RF atoms.

  Each actuator identifier has an implicit boolean return value which corresponds to the success or failure of that actuator. This success value is provided by a direct mapping to the external actuator code.

- **PRE** – PRE denotes what the agent must believe for the actuator code to be invoked. These pre-conditions are a belief query sentence. If the agent's Belief Set does not entail these pre-conditions then the actuator should not be invoked.

- **POST** – POST defines what should be true after the actuator has completed. This post-condition is a disjunction of positive belief statements, one of which should be added to the agent's belief state as a result of invoking the actuator,

under the required pre-conditions. The ALPHA interpreter does not do the job of deciding which outcome has occurred as a result of running the actuator; Instead, it is the responsibility of the designer of the actuator to ensure that the relevant beliefs are adopted.

In designing ALPHA, we acknowledge that little in real life programming is guaranteed. Therefore, the actual post-condition of the actuator is assumed to be weaker than the POST statement. The post-condition will implicitly include a BELIEF(TRUE) to recognise that any actuator can potentially fail to produce the outcome as specified by the actuator designer in the POST statement.

- **CODE** – CODE explicitly defines a piece of *external code* which is invoked to perform the action. This effectively make an actuator something similar to a native method call in Java. As with perceptors, multiple CODE declarations can be defined, but only one piece of code is used on any platform. Actuators can add new beliefs to the agent's belief set through defined pipes into the belief set. Actuators in Java and C are supported directly by ALPHA, while many other languages and applications can be supported if they have an exportable C interface.

  External actuator code returns a boolean value corresponding to the success or failure of the actuator. This value is mapped directly to the ALPHA actuator identifier, providing a success or failure value for the actuator as a whole. The arity of the actuator implementation must match the arity of the action itself. All actuator arguments are passed directly to the external actuator code as a single RF statement that includes the actuator name as functor.

The explicit definition of PRE and POST conditions in the ACTION construct allow for means-end reasoning to be performed in the Goal achievement process (See Section 9.4 for examples of use).

The AF-APL actuator definition `ACTUATOR ie.ucd.invokedtor;` are not valid in ALPHA. Instead, this *short hand notation* is still permitted in role files, but is interpreted by the ALPHA compiler as an actuator definition file that is to be included.

Actuators can only return true or false values – indicating if the actuator was successful or not. However, during execution, an actuator can add beliefs to the agent's belief set.

5.2 Actuator Execution

An actuator corresponds to one action which the agent can perform in its environment. The action itself need not be simple; For example `eat(food)` could be quite a complex action in itself. Nevertheless, it can be convenient to build even more complex procedures out of individual actions. The PLAN operations, introduced in the next chapter, allow more complex actions to be built out of simple actuators.

There is no main() for an agent, that it there is no single point from which actuators are executed in some sequence. Instead, an agent can either commit to invoking an actuator (or plan) or can reactively invoke an actuator in response to some external stimulus. Commitments and reactive rules are discussed further in the following chapters.

5.3 Uniqueness of Actuators

For each actuator held by an agent, one, and only one, instance of the actuator code is instantiated. This means that although the same actuator can be invoked in parallel, only one instance of the actuator code is ever utilised. The implications of this is that either: (a) actuators code must be thread safe; or (b) the ALPHA code must be designed to be thread safe when non thread safe actuators are being used.

Each agent has one instance of each actuator used by that agent; That is, the the interpreter does not have one central store of actuators, with different agents using the same actuator code in parallel. An actuator will be instantiated once and once only for each agent. If two agents both make use of the same actuator on the same platform, then two instantiations of that actuator will be made (one for each agent). Each agent will then keep reference to their instantiation of their actuator and use it when appropriate.

5.4 Altering the Mental State of the Agent

For a number of reasons, it is necessary for an agent to have the ability to alter its own mental state through actuators. The most obvious need is to give the agent a form of 'a priori perception'; that is, to allow the agent to adopt beliefs as a result of executing an action. Actuators also need to alter mental state in order to allow the agent to learn new tasks, skills, and relationships.

In implementation, actuators alter the mental state of the agent through a number of mental state operations. In Java, these operations are supplied as a number of methods inherited from a basic Actuator class; While in C, a library of functions is provided to achieve the same goal.

For an ALPHA agent to have intelligence in terms of ability acquire new knowledge and skills, it is essential that the agent be capable of altering its own mental state at run-time. This capability does however come at the price of safety and predictability in the agent design; For example, there is nothing to stop an actuator being written, which inadvertently deletes the whole mental state of the agent at run-time. An alternative approach to the model presented here, would be to prohibit the alteration of anything other than the agent's beliefs.

5.5 Behavioural Actuators

The basic actuator model is seen as something akin to a function, in that it is assumed to terminate eventually. The use of the `BEHAVIOUR` keyword on an actuator identifier can be used to indicate that this is not a self terminating actuator. Instead, these actuators are assumed to proceed until they are explicitly terminated by the interpreter. This can be useful in robotics for allowing behaviours such as wall following to proceed until some belief becomes true.

Actuators must be defined as BEHAVIOUR to be used with the DOUNTIL plan operator. Similarly, actuators which have been defined as BEHAVIOUR cannot be used

outside of a DOUNTIL operator.

Chapter 6

Plans

Plans provide a mechanism for an agent's actions to be arranged at a high level. ALPHA allows plans to be statically provided at design time, or dynamically generated at runtime by the goal directed, means-end reasoning system. This chapter introduces the format of ALPHA plans – specifying the basic plan construct and each of the planning operators that can be used for manipulating beliefs.

6.1 The PLAN Construct

The PLAN construct explicitly defines a plan that can be used to perform a complex action. The plan construct takes the the following form:

```
BEGIN{PLAN}
IDENTIFIER ie.ucd.myProject.myPlan(?x);
PRE BELIEF(TRUE);
POST BELIEF(i_did_things);
BODY SEQ(doThis,doThat,doTheother(?x));
END{PLAN}
```

The PLAN construct is very similar to the ACTUATOR construct and contains the following sections:

- **IDENTIFIER** – The plan's IDENTIFIER uniquely defines the plan within the ALPHA namespace, and explicitly defines the arity – the number of arguments – taken by the plan. Arguments supplied to the PLAN construct must be RF atoms. Each plan identifier has an implicit boolean return value that corresponds to the success or failure of that plan. This success value is provided by a direct mapping to the result of *executing* the plan body.
- **PRE** – PRE denotes what the agent must believe for the plan body to be invoked. This pre-conditions is a belief query sentence. If the agent's Belief Set does not entail this pre-condition, then the plan body will not be invoked.

- **POST** – POST defines what should be true after the plan has completed. This post-condition is a disjunction of positive belief statements, one of which should be added to the agent's belief state as a result of invoking the plan under the required pre-conditions. The ALPHA interpreter does not do the job of deciding which outcome has occurred as a result of running the plan; Instead, it is the responsibility of the developer to ensure that the relevant beliefs are adopted.

  In designing ALPHA, we acknowledge that little in real life programming is guaranteed. Therefore, the actual post-condition of the plan is assumed to be weaker than that specified with the POST statement. The post-condition will implicitly include a BELIEF(TRUE) to recognise that any plan can potentially fail to produce the outcome as specified by the plan designer in the POST statement.

- **BODY** – BODY defines a construction of actuator and plans that should be invoked to achieve the required plan. Any free variables in the plan body – excluding those scoped by a FOREACH construct – must be scoped by one of the arguments to the PLAN construct.

  Activities used in a plan BODY must match a plan or actuator included in the agent design. Recursive calling of a plan is allowed.

The explicit definition of PRE and POST conditions in the PLAN construct allow for means-end reasoning to be performed in the Goal achievement process (See Section 9.4 for details of this process).

6.2 The Arrangement Constructs

In chapter 5 we introduced the actuator as the basic unit of action which can be executed by the agent. To build more complicated actions out of simple actuators, we may use plan constructs to sequence, order and test the mental state of the agent. In this section we introduce ALPHA's plan constructs. They may be used directly within a commitment declaration (COMMIT), or may be used in the definition of plan objects as described above.

6.2.1 SEQ – The Sequence Construct

SEQ defines a set of steps that must be realised in the order listed. For example, consider the actions that are performed to boil water for a cup of tea: (a) get a kettle; (b) fill the kettle; and (c) boil the kettle. If actions – or indeed other plans – exist to perform these steps, then we can use the SEQ construct to build an appropriate plan body:

```
SEQ(getKettle,fillKettle,boilKettle)
```

SEQ can operate on any whole number of arguments – including just one. The case of operating on one argument is not particularly interesting; the SEQ construct is said to succeed if its one argument succeeds, and fails otherwise. The same is true for each of the other *arrangement* constructs: PAR, AND, OR, and XOR.

6.2.2 PAR – The Parallel Construct

PAR defines a set of steps that may be realised simultaneously. Looking at the tea example again, let us consider the situation of adding milk and stirring. These two actions can often be performed in parallel; Therefore, if suitable actions exist, we can use the PAR construct to build a plan body to perform these two actions in parallel:

```
PAR(stir,addMilk)
```

When using the PAR construct, you are implying that there is a clear preference for all actions to be performed in parallel. If the requirement is not very strict, then the AND construct should probably be used.

6.2.3 AND – The Random Order Construct

AND defines a set of steps which can be realised in any order, parallel or sequential. All steps must be performed, but it really does not matter what order the steps are performed in; as long as each step is eventually achieved, it does not matter how they are sequenced. Coming back to the tea example: consider the actions of getting tea, getting a cup, and getting the milk. There is no particular order evident here; Therefore, we can use the AND construct as follows:

```
AND(getTea,getCup,getMilk)
```

By default, the interprete should simply try to sequence the actions as if a SEQ construct was used. However, if an action fails initially, the next action will then be attempted, and the agent will then come back to the failed action. This cycle will continue until all actions have been achieved, or remaining sequence of actions are attempted, each of which fail in a row.

6.2.4 OR – The Non-Deterministic Choice Construct

OR defines a set of steps, one of which must be realised. All steps are tried in parallel. Once one of these steps is realised, the construct is said to have succeeded; Thus resulting in all other steps being abandoned. If none of the steps return true, then the construct is said to have failed. Using the tea example again, it is of course possible to boil water in more than one way. In addition to using a kettle, water may also be boiled in a pot. This results in two alternative ways to get boiling water, which can be expressed in a plan body as:

```
OR(SEQ(getKettle,fillKettle,boilKettle),
   SEQ(getPot,fillPot,boilPot)
  )
```

The OR construct is useful when we wish to try out many different options as the same time. However, in practice, it is not always useful to perform all operations together.

In our boiling water example, it is probably a bad idea to both try to get boiling water from the pot and from the kettle at the same time. Instead, we often try each option in turn; To do this we use the XOR construct.

6.2.5 XOR – The Deterministic Choice Construct

XOR defines a set of steps, which must be performed in the order presented, and which succeeds when one of the steps is realised. As with the OR constructor, if any one step succeeds, OR is said to succeed; However, if all steps fail, OR also fails. Our boiling water example can be expressed with XOR:

```
XOR(SEQ(getKettle,fillKettle,boilKettle),
    SEQ(getPot,fillPot,boilPot)
   )
```

As we have seen in in the bailing water example, it is possible to nest OR, XOR, AND, SEQ, and PAR constructs. This allows us to build up a more complete plan body for making a cup of tea:

```
SEQ(AND(getTea,getCup,getMilk),
    XOR(SEQ(getKettle,fillKettle,boilKettle),
        SEQ(getPot,fillPot,boilPot)
        ),
    AND(addTea,pourWate)r,
    PAR(stir,addMilk)),
    tasteTea
    )
```

6.3 The Mental Constructs

These are a number of in-built mental constructs that allow an agent to perform explicit actions on its own mental state. These constructs are defined as part of the ALPHA language rather than as actuators.

6.3.1 The COMMIT Construct

Chapter 7 introduces commitments and commitment management as the core of the ALPHA processing model. Details of the notion of a commitment, and its treatment are omitted until the next chapter; Here we merely introduce the syntax of the COMMIT construct which will be used for commitment adoption. The construct takes four arguments, which are in turn:

- **Agent Name** – The agent to which the Commitment is to be made.
- **Time** – The earliest time at which the agent is to attempt to achieve the commitment.

- **Maintenence Condition** – The maintenence condition of the commitment. This is a belief query sentence which must be true as long as the agent is to hold the commitment. If, at any time after adopting the commitment, this Maintenence Condition is no longer held, then the Commitment will be dropped.
- **Activity** – An identifier of the plan or action to which the agent is committing. The identifier can also be a plan body directly.

Some example COMMIT constructs include:

- `COMMIT(Anna,20.00,BELIEF(TRUE),eat(Dinner))`

  This represents a commitment made by the agent to another agent *Anna*to have dinner at 20.00. In this case, we assume that eat is an identifier which matches either a plan or action identifier that has been defined in the ALPHA code. The maintenence condition is BELIEF(TRUE), and since an agent always believes *TRUE*, there are no circumstances under which the COMMITMENT will be dropped prior to either the successful or failed execution of haveDinner.

- `COMMIT(?Self,21.00,!BELIEF(have_lots_of_work),watchTV)`

  Here a commitment is being made by an agent to itself to watch TV at 21.00. *?Self* is a key variable defined in ALPHA. When a commitment is adopted based on such a COMMIT structure, the ?Self variable will be replaced by the name of the agent which is adopting the commitment. The result of the maintenence condition here is that if the Agent believes that he is busy, then he will drop the commitment immediately.

- `COMMIT(Anna,20.00,BELIEF(TRUE),SEQ(gotoShops,buyFood,cookDinner))`

  Here the agent has made a commitment with typical Agent, Time and Maintenence condition values. However this time rather than committing to a plan or action with a given identifier, the agent will instead commit directly to a plan body.

- `COMMIT(Rem,+00:10,BELIEF(TRUE),meetForCoffee)`

  Here we see that different forms of Time identifiers are allowed within ALPHA. Rather than specifying an absolute time such as 20.00, we see here that a relative time can be specified. In this example the agent will first attempt to achieve the new commitment 10 minutes after adopting the commitment.

- `COMMIT(Rem,?Now,BELIEF(TRUE),meetForCoffee)`

  In this example we see that *?Now*is an ALPHA keyword for the current time. Semantically, *?Now*is s synonym for *+0*. Hence the agent will attempt to achieve the commitment immediately after adopting the commitment.

Although the COMMIT construct is typically only seen in the declaration of a commitment rule, the construct can be nested within other plan bodies. For example, in a complicated plan body, we can declare details of the style of secondary commitment to be adopted; overriding the default provided by the commitment management process. For example, we can explicitly state that the earliest start time of some action, doE, should be a minimum of ten minutes after the predecessor action with the following construction:

```
SEQ(doA,
    PAR(doB,doC),
    SEQ(doD,
        COMMIT(?parent,+10,BELIEF(TRUE),doE)
        doF
       )
    )
```

6.3.2 The ADOPT_GOAL Construct

In ALPHA a goal is viewed as a a state of the world – or set of beliefs – that an agent wishes to bring about. Goals may be adopted using either the ADOPT_GOAL or ACHIEVE_GOAL constructs; ADOPT_GOAL adds a goal to the agent's goal set, returning immediately; while ACHIEVE_GOAL adds the goal to the agent's mental state, but does not return until the GOAL is achieved. For example, we can represent that the agent is to adopt a goal to keep the door closed as follows:

```
ADOPT_GOAL(closed(door))
```

A goal cannot be added to the agent's goal set if that goal has already been met (the goal is already entailed by the agent's belief set); in such cases the ADOPT_GOAL construct will still return true, reflecting a momentary adopting of the goal by the agent before its immediate natural fulfilment.

6.3.3 The ACHIEVE_GOAL Construct

Like ACHIEVE_GOAL, ADOPT_GOAL adds a goal to the agent's belief set, but differs in that it does not return until the goal is achieved by the agent. If the goal is already entailed by the agent's belief set, the ACHIEVE_GOAL construct returns immediately. The process used to achieve goals is discussed further in section 9.

6.3.4 The ADOPT_BELIEF Construct

This construct takes one argument, a BELIEF to be added to the agent's belief set. This is equivalent to the defunct adoptBelief actuator. The actuator always returns successful, if it was given a well defined belief statement. For example, to add a belief that Rem is hungry to the agent's belief set, the following construct can be used:

```
ADOPT_BELIEF(BELIEF(hungry(Rem)))
```

6.4 The Imperative Constructs

The following constructs provide basic tools for the development of complex plan descriptions to be used by agents. Some of the constructs are expanded at compile time into

structures of more primitive constructs, while others are processed explicitly through commitment management at runtime.

6.4.1 TEST – The Belief Query Construct

The TEST construct allows us to test if particular beliefs are held by the agent. The construct takes one argument, a belief query sentence, which may or may not contain free variables. If the agent's mental state does entail the belief query sentence, then the construct is said to succeed. Using the TEST construct is equivalent to creating an actuator which explicitly checks the mental state of the agent; the construct is however defined as part of the language; meaning it is used more efficiently than an external actuator. We can use TEST to decide if we want to drink the tea that we have made:

```
SEQ(tasteTea,
    XOR( SEQ(TEST(BELIEF(tea_tastes_good)),
             enjoyTea
            ),
         SEQ(TEST(BELIEF(tea_tastes_dodgy)),
             drinkItAnyway
             ),
         drinkCoke,
         )
    )
```

When TEST is used within a PAR construct the test must hold true throughout the evaluation time of the PAR construct, and not simply at any one time during the evaluation. This behaviour allows a convenient way of ensuring that some mental state holds throughout a course of actions.

Since there may be multiple values possible for any one query of the belief set, TEST cannot be used to set individual values. Instead, the FOREACH construct must be used, forcing designers to consider the possibility of multiple matches.

6.4.2 FOREACH – The Universal Quantification Construct

FOREACH allows an agent to check the contents of its mental state and assign variables into plans based on this mental state. FOREACH takes two arguments: a belief query sentence, and a plan body with free variables (all of which must be potentially scoped by the belief query sentence). To illustrate consider an agent that wants to hold a party, and therefore wishes to invite all its friends to the party; we can express this with the FOREACH construct as follows:

```
FOREACH(BELIEF(friend(?name)),
        AND(invite(?name))
        )
```

At runtime, if the agent's belief set includes the beliefs that it has friends: Anne, Jane and Freddy, then the above statement will be expanded to:

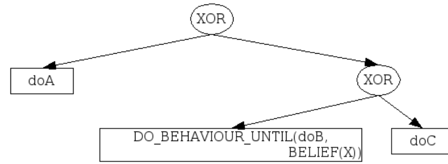


Figure 6.1:

```
AND(invite(Anne),
    invite(Jane),
    invite(Freddy)
    )
```

The plan body to be expanded must: (a) be based on an arrangement construct e.g. XOR, OR, AND, PAR or SEQ; (b) contain only one argument.

If the belief query sentence fails – in this case, because the agent has no friends – then the FOREACH construct is said to fail.

It is possible to use the FOREACH construct to substitute actions as well as arguments into the plan body. For example, the mental state of the agent can be checked for actions to be performed as follows:

```
FOREACH(BELIEF(actionToTry(?action)),
        AND(?action)
        )
```

6.4.3 The DO_BEHAVIOUR UNTIL Construct

The DO_BEHAVIOUR_UNTIL construct is used to execute behaviour type actuators. The construct takes two arguments: (a) the identifier of a behaviour type actuator to be executed; and (b) a belief query sentence, which can contain free variables. The actuator is invoked and will be left to run until it returns, or until the agent's belief set entails the query – in which case, the actuator is forcefully stopped. To illustrate, consider a robot which is to move along a wall until it finds a door:

```
DO_BEHAVIOUR followWall UNTIL BELIEF(found(Door))
```

To be used in this construct, the followWall actuator must have been declared as a BEHAVIOUR actuator.

6.4.4 The TRY RECOVER Construct

The TRY_RECOVER plan construct is intended for more than simply graceful degradation of plans – recovery plans used with the construct should attempt to remedy a plan in situ. This approach reflects our belief that it is often very wasteful for an agent to give

up on a plan, and that agents should put more effort into recovering from unpredicted situations. The approach taken here is similar to the *Interrupt Mechanism* introduced to 3APL in [14], but produces a simpler implementation model which does not effect the semantics of the underlying logic.

TRY_RECOVERY is used to indicate whether an action – or indeed a complete plan structure – should have error recovery mechanisms associated. The construct takes two arguments: first, the action or plan body to be monitored for failure; second, the plan which is to be used for recovery. Unlike the *try catch* exception handling mechanisms in Java, TRY_RECOVER attempts to repair an erroneous situation and return the agent to finish the original action. Therefore it is possible to re-write any TRY_CATCH construction in a more basic form, by simply making the following substitution for each action to be monitored:

```
act_ident -->
   XOR(?act_ident,
       SEQ(?recovery_plan(?act_ident),
           ?act_ident))
```

This replacement allows an agent to attempt an action. If the action fails, the agent will then try the recovery of the situation; this failing, the action in general fails. However, if the recovery plan succeeds, then the agent will attempt the original action once again. Failure on this occasion leads to complete failure of the action. To illustrate, the plan body SEQ(doA,doB) can be tagged for recovery with the *social_recovery* plan as follows:

```
TRY_RECOVER(SEQ(doA,doB),
            social_recovery)
```

Using the substitution above, this statement can be expanded at runtime to this simpler representation:

```
SEQ(XOR(doA,
        SEQ(social_recovery,doA)),
    XOR(doB,
        SEQ(social_recovery,doB))
    )
```

This ALPHA construct allows any number of recovery plans to be defined and used.

6.5 Plan Execution

Plans define an arrangement of actions to allow an agent to achieve an implicit or explicit goal. Plans give an order of action execution, contingencies for the failure of a plan branch, and the ability to make choices based on the mental state of the agent.

Plans are however open to some forms of interpretation. For example, even in imperative programming, we rarely explicitly define when each step of a plan should be executed.

ALPHA acknowledges that there are different ways to achieve any given plan; different strategies that can be used to achieve each step of the plan. Chapter 8 describes how the agent may try to achieve a well specified plan in a dumb but predicatable way. Oh the other hand, the commitment manager, described in the next chapter, breaks each step of the plan down into commitments to action, The commitment management is a complicated system, which tries to take advantage of the agent's entire mental state, and other agents in its environment, in order to achieve the goal. Through its more complex design, the commitment manager has a better hope of achieving the plan as specified.

Chapter 7

Commitments & Commitment Rules

A commitment is a promise made by an ALPHA agent (to itself or other agent) to attempt an action. The commitment lies at the heart of the ALPHA execution model, and is essential to giving ALPHA agents some inherent intelligence. This chapter introduces commitments: showing their form; describing how they become adopted; and detailing their use in the agent's rational execution model.

A commitment is a promise made by an ALPHA agent (to itself or other agent) to attempt an action. In other words, a commitment is the mental equivalent of a contract. As such, it specifies the course of action that the agent has agreed to, to whom this agreement has been made, when it must be fulfilled, and under what conditions the agreed course of action becomes invalid (i.e. under what conditions the contract can be breached). For example, we can represent that Rem has committed himself to eat biscuits at 11am (as long as he believes that he has no lunch plans) as follows:

```
COMMIT(Rem,
       11:00,
       !BELIEF(has(lunch_plans)),
       eat(biscuits)
       )
```

The rational realisation of commitments is at the heart of ALPHA's execution and reasoning model. An ALPHA agent may have any number of commitments at any time; These – potentially contradictory – commitments, stored in the agent's commitment set, are normally acquired by the agent in a number of ways. First, initial commitments may be defined explicitly in ALPHA code; Second, Commitment Rules allow an agent to adopt commitments if its belief set entails a condition; Third, commitment management produces new commitments by refining more primitive commitments through the application of plans and reasoning; And finally, goals management produces commitments to action that can achieve a given goal. The goal management process is discussed in chapter 9. Section 7.1.2 introduces the commitment rule; followed in section 7.3 with

a detailed description of the commitment management process which is responsible for the refinement and management of commitment structures. ALPHA agents can reason directly about their own commitment set; this reasoning capability is described in section 7.4. First however, we look at primary commitment adoption.

7.1 Primary Commitment Adoption

7.1.1 Initial Commitments

At its start time, an agent can have any number of initial commitments in its commitment set. Such a commitment, once adopted, is known as a primary commitment since it has no parent. Initial commitments are defined with the COMMIT construct (See section 6.3.1). For example, consider an agent that has an actuator (say(?x)); then the agent can be designed to have an initial commitment to say *hello* immediately, through the inclusion of the following statement:

```
COMMIT(?Self,?Now,BELIEF(TRUE),say("hello"));
```

7.1.2 Commitment Rules

An agent's may also adopt primary commitments by means of commitment rules. The commitment rule allows an agent to rationally decide on a course of action based on its mental state. Take, for example, an agent who is hungry and has a piece of fruit. Under these circumstances, a rational action for the agent to take, would be to eat the fruit. We can specify this as a commitment rule in ALPHA with the following:

```
BELIEF(hungry(me))
& BELIEF(haveFruite(?someFruit))
  => COMMIT(?Self,
            ?now,
            BELIEF(TRUE),
            eat(?someFruit)
            );
```

The commitment rule has two distinct parts:

- Belief Query Sentence – A belief query sentence which may or may not contain free variables. When the agent's Belief Set entails this sentence, the Commitment Rule is said to be triggered, resulting in the commitment being adopted, or added, to the agent's commitment set.
- Commitment – A COMMIT construct defines a new commitment to be adopted if, and only if, the adoption condition is entailed by the agent's Belief Set. Please see section 6.3.1 for details on the COMMIT construct. Any free variables within the COMMIT construct must be entailed either by ALPHA key terms (?Self,?Now) or a substitution set generated from the Unification of the Belief Query Sentence with the agent's Belief Set.

The simplist commitment form is for an agent to commit to executing an actuator. To illustrate, consider an agent that has an actuator (`eat(?food)`) that can be used to eat an apple; Then, if the agent believes that it is hungry and has an apple, the following commitment rule can cause the agent to immediately commit to eating that apple:

```
 BELIEF(hungry) & BELIEF(have(apple)) =>
                    COMMIT(?Self,
                           ?now,
                           BELIEF(TRUE),
                           eat(apple));
```

As well as committing to individual actuators, an agent may also commit to plan bodies. Therefore, if the agent believes that it wants pizza, it can commit to a simple plan to order and collect a pizza with the following commitment rule:

```
 BELIEF(want_pizza(?Self)) & BELIEF(favourite_pizza(?type)) =>
                COMMIT(?Self,
                       ?now,
                       BELIEF(TRUE),
                       SEQ(order_pizza(?type),
                           collect_pizza,
                           eat(pizza)
                          )
                      );
```

However, if the agent tries to collect the pizza immediately after ordering it, he is left waiting for at least ten minutes. Therefore, we can use a nested COMMIT construct to advise that the action of `collect_pizza` should not be first attempted until at least 10 minutes after the order was placed.

```
BELIEF(want_pizza(?Self)) & BELIEF(favourite_pizza(?type)) =>
              COMMIT(?Self,
                     ?now,
                     BELIEF(TRUE),
                     SEQ(order_pizza(?type),
                         COMMIT(?self,+10,BELIEF(TRUE),collect_pizza),
                         eat(pizza)
                        )
                    );
```

Agents may also commit to executing pre-compiled plans. Therefore, if our agent has a plan, havePizza(?type), with a plan body matching that above (ordering the pizza, collecting the pizza, and eating the pizza), then the commitment rule above can simply be re-written as:

```
BELIEF(want_pizza(?Self)) & BELIEF(favourite_pizza(?type)) =>
```

```
                 COMMIT(?Self,
                        ?now,
                        BELIEF(TRUE),
                        havePizza(?type)
                       );
```

Commitment rules may be used to commit the agent to many similar tasks in parallel. For example, consider an agent with a dialog plan askOutForDrink(?x), and with a belief set containing both BELIEF(friend(Anna)) and BELIEF(friend(Derry))); Then, if the agent believes that it is thirsty, the following commitment rule will cause the agent to ask each of its friends out for drinks in parallel.

```
BELIEF(thirsty) & BELIEF(friend(?buddy)) =>
                 COMMIT(?Self,
                         ?now,
                         BELIEF(TRUE),
                         askOutForDrink(?buddy));
```

All free variables used on the right hand side of a commitment rule must be bound by variables on the left hand side of the commitment rule (or through the use of constructs such as FOREACH); therefore the following is an invalid commitment rule:

```
BELIEF(thirsty) & BELIEF(friend(?buddy)) =>
                 COMMIT(?Self,
                         ?now,
                         BELIEF(TRUE),
                         askOutForDrink(?friend));
```

since ?friend cannot be instantiated.

Ordering of Commitment Rules

There is an implicit ordering on commitment rules provided by their position in ALPHA code.

7.2 The Commitment Set

7.2.1 Commitment Structures

In the last section we saw how initial commitments and commitment rules can be used to add primary commitments into an agent's commitment set. If we consider a commitment to perform some sequence of actions – let us call it a complex commitment – then we can break this complex commitment down into a structure of more elementary commitments. For example, take the commitment to order, collect, and eat a pizza; A single complex

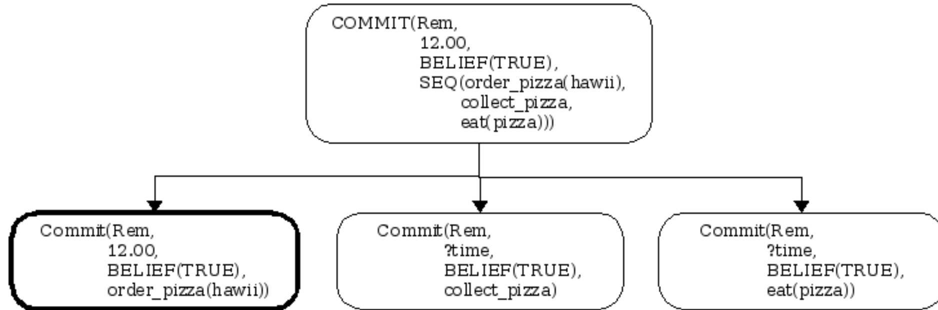


Figure 7.1:

commitment can be broken down into a structure of more primitive – or secondary – commitments as shown in figure 7.1.

These commitment structures are stored in the agent's commitment set. where they evolve over time, reflecting their partial processing. The processes which lead to he commitment structures evolving are detailed in the following sections. One of the most important concepts in this evolution process are *active commitments* within the commitment structure. A commitment structure can have a number of active commitments at any time; these are the commitments which are being actively achieved by the agent. In our pizza example, the first step to be performed is ordering the pizza; in the diagram this has been marked as the active commitment with a dark border.

7.2.2 Ordering of the Commitment Set

During any one agent execution cycles, each commitment structure in the agent's commitment set, is selected once and once only for refinement; the ordering on commitment structures is temporal.

7.3 Commitment Structure Refinement

Commitment refinement is a stepwise process whereby commitment structures are expanded and contracted, invoking low level actuators to achieve the agents implicit or explicit goals.

The approach taken to commitment structure expansion is to leave it until the last possible moment – that is, when the commitment becomes active. This has the advantage that these constructs are expanded out with the most relevant information – essential for a FOREACH construct. On the downside, the agent cannot reason into commitment structures beyond a certain depth. We will call this 'The Oracle Constraint' – the agent cannot reason past a choice he does not fully understand; however, we will understand the choice when we come to it.

7.3.1 Refining Commitments to Primitives

Identifying Primitive Commitments

A commitment to an action identifier – called a primitive commitment – is a commitment that has been made to some actuator or plan identifier. The action committed to is identified as being a primitive if, and only if, the operand is not recognised as one of the planning constructs as specified in the last chapter. For example, a commitment by Rem to eat an apple at 12.00 takes the form:

```
COMMIT(Rem,12.00,BELIEF(TRUE),eat(Apple))
```

Evaluating Maintenence Conditions

During each execution cycle, each maintenence commitment's maintenence condition is checked for validity. If the maintenence condition is no longer met, then the commitment, along with any child commitments, are dropped. Any parents are also notified of the *failure* of the commitment.

Resolving Primitive Commitments

When a primitive commitment is encountered, the action committed to must be resolved to an appropriate actuator or plan body. Since actuators are the most primitive action, they are first to be resolved against, then followed by the plan set. Actuators have an ordering based firstly on their position in ALPHA code, and secondly, within each actuator there is a similar ordering on CODE implementation blocks. Therefore if a commitment to `eat(apple)` was being processed, and the agent held the following two actuator constructs (in this order):

```
BEGIN{ACTUATOR}
IDENTIFIER ie.ucd.myProject.eat(?food);
PRE BELIEF(TRUE);
POST BELIEF(i_did(x)) | BELIEF(i_did(y);
CODE ie.ucd.myproject2.actuators.Actuator.class;
END{ACTUATOR}

BEGIN{ACTUATOR}
IDENTIFIER ie.ucd.myProject2.eat(?food));
PRE BELIEF(TRUE);
POST BELIEF(i_did(x)) | BELIEF(i_did(y);
CODE ie.ucd.myproject2.actuators.Actuator.class;
CODE ie.ucd.myproject2.actuators.Actuator.o;
END{ACTUATOR}
```

then the ie.ucd.myproject2.actuators.Actuator.class implementation would first be tried.

As with actuators, there is an ordering on plans based on their position in ALPHA code; meaning that when an identifier is encountered, it is tested against the first plan construct, then any other plan construct.

Firing Primitive Commitments

Once an actuator has been selected, the actuator is *fired* by calling that actuator and passing the appropriate arguments. Actuators are fired in their own threads – guaranteeing that ALPHA agents never become locked because of badly designed external actuator code.

Once the actuator fires, it is marked as active and remains so until the actuator code returns. The commitment will not be treated any further in the present commitment cycle.

Primitive commitments to plan identifiers are *fired* by expanding the plan body out and assigning a new active commitment within that commitment structure.

Each child commitment must have a particular minimum start time, maintenence condition, and agent to whom the commitment is being made. By default, secondary commitments inherit the commitment owner from their parents, have a minimum start time, which reflects the earliest possible time the action can be initiated, and a single minded maintenence condition. The single minded maintenence

Weak minded commitment management treats each commitment in the traditional *Open-Minded* way. However, secondary commitments are created with a maintenence condition which says that the commitment will be attempted *once and once only* . Thus, a secondary commitment is weak, meaning that the commitment manager will attempt the commitment once and then drop the commitment whether or not it was successfully achieved. Primary commitments are still created with maintenence conditions as specified in the commitment rule, meaning that they are still treated in the traditional *Open Minded* way.

Dropping Primitive Commitments

If a commitment to an actuator is active, then the return value is checked. If the return value has not been set yet, the commitment to the actuator is ignored until the next agent execution cycle. If the return value has been set to *true* , then the commitment is said to be fulfilled, and is hence dropped, with any relevant notifications made to ancestral commitments. However, if the actuator has been found to return false, then the commitment to the actuator is re-set, to be processed as new during the next execution cycle.

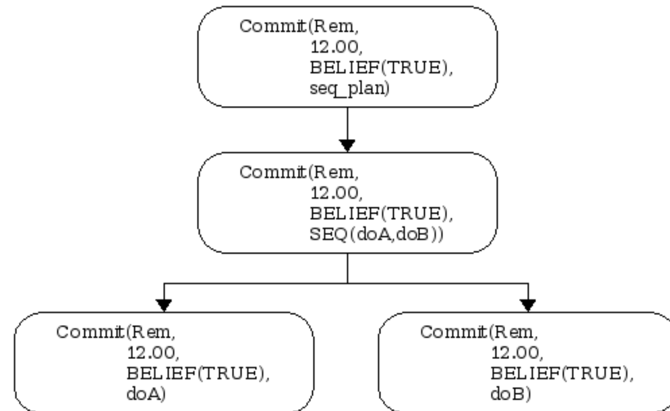
Putting it all together

In summary, the algorithm for processing primitive commitments is shown in figure 7.2.

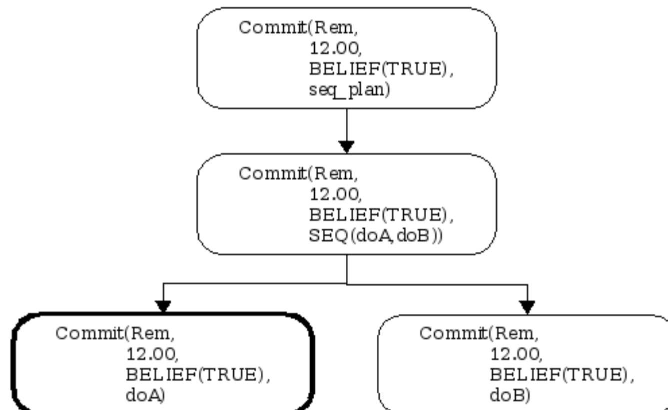
```
processPrimitiveCommitment(commitment)
{
   if the commitment's maintanence condition still holds
   {
      if the commitment is inactive
      {
         if the minimum start conditions have been met (time + sibs)
         {
            if actuator can be found
               fire actuator it its own thread
            else if plan can be found
               expand commitment structure with plan
               unmark this commitment as active
               mark the new (relevant) children as active
         }
         else
         {
            do nothing
         }
      }
      else if the commitment is active
      {
         if a retrun value has come from the actuator
         {
            if the return value is true
            {
               commitment is fulfilled
               drop commitment
               notify any ancestors
            }
            else if the return value was false
            {
               mark commitment as inactive
            }
         }
         else if no return value has come in already
         {
            do nothing
         }
      }
   }
   else if the maintanence condition no longer holds
   {
      Drop the primitive commitment
   }
}
```

Figure 7.2:

Step 1 (Start) – Primary Commitment Adopted and Active

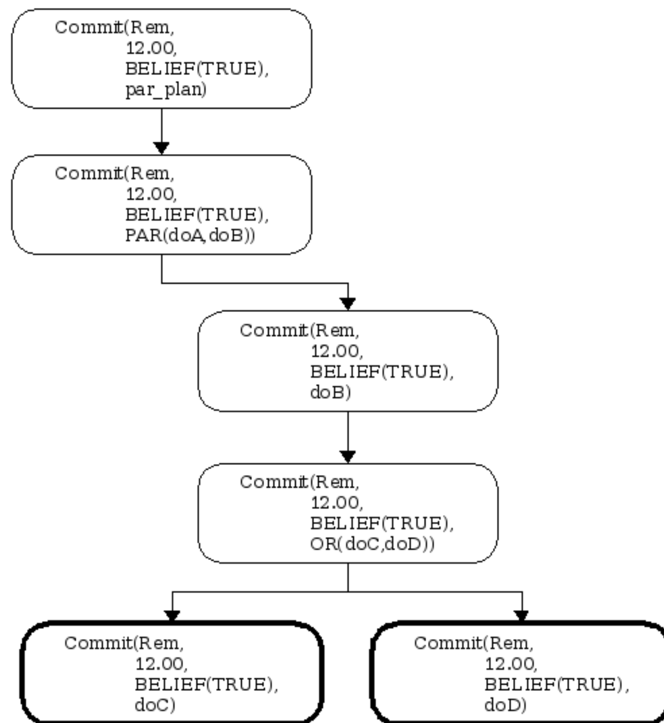


Step 1 (End) – Primitive Commitment resolved against plan; plan body used to expand commitment structure.



Step 2 (End) – Commitment to doA made active. doA actuator invoked.

Figure 7.3:



Step 3 – doA returns true, hence commitment to doA fulfilled and dropped. Commitment to doB now becomes active. doB resolved to plan body and used to expand commitment structure. doC and doD become active and both actuators are invoked.

Step 4 – doD actuator returns true, hence the commitment to doD is fulfilled and dropped. Subsequently commitment to doC is redundant and dropped. Parent commitments are fulfilled and are also dropped.

Figure 7.4:

7.3.2 Refining Commitments to Imperative Constructs

The TRY_RECOVER constructs is replaced with structures of more primitive constructs at compile time; Thus, they do not need to be considered here. The only construct which directly effects the building of commitment structures, is the FOREACH construct, which is now discussed.

FOREACH

The FOREACH construct is used for expanding out constructs based on the beliefs held by the agent. When a FOREACH construct becomes active, it explicitly alters the commitment structure. Figures 7.5 and 7.6 present an example of the successful application of a FOREACH construct. Until the construct becomes active, it is left in its compacted state. When it becomes active, the test condition is evaluated, and any variables returned are used to instantiate a new commitment structure below the commitment to FOREACH. However, if the FOREACH test does not return any positive values, the commitment to FOREACH is assumed to have failed. Normal maintenence condition evaluation determines if the commitment is dropped.

DO_BEHAVIOUR_UNTIL

Commitments to DO_BEHAVIOUR_UNTIL are staticly bound to one argument, the actuator which must have been declared as behavioural; the commitment is always a leaf commitment in a commitment structure.

7.3.3 Refining Commitments to Mental Constructs

The ACHIEVE_GOAL, UPDATE_GOAL, ADOPT_BELIEF constructs are treated as actuators by the belief refinement process; that is, these refinement of these commitments does not explicitly alter the commitment structure, although the goal constructs do indirectly cause new commitment structures to be generated, which is discussed further in the next chapter. The only mental construct which does cause alteration of the current commitment structure is the COMMIT construct which is now discussed.

COMMIT

The COMMIT planning construct is used to override the default secondary commitment construction semantics.

7.4 Reasoning about Commitments

Since an ALPHA agent's mental state includes its commitments, an agent can reason on its own commitments. The commitment construct is similar to the commit construct introduced earlier. However, the COMMIT construct is used to adopt new commitments,

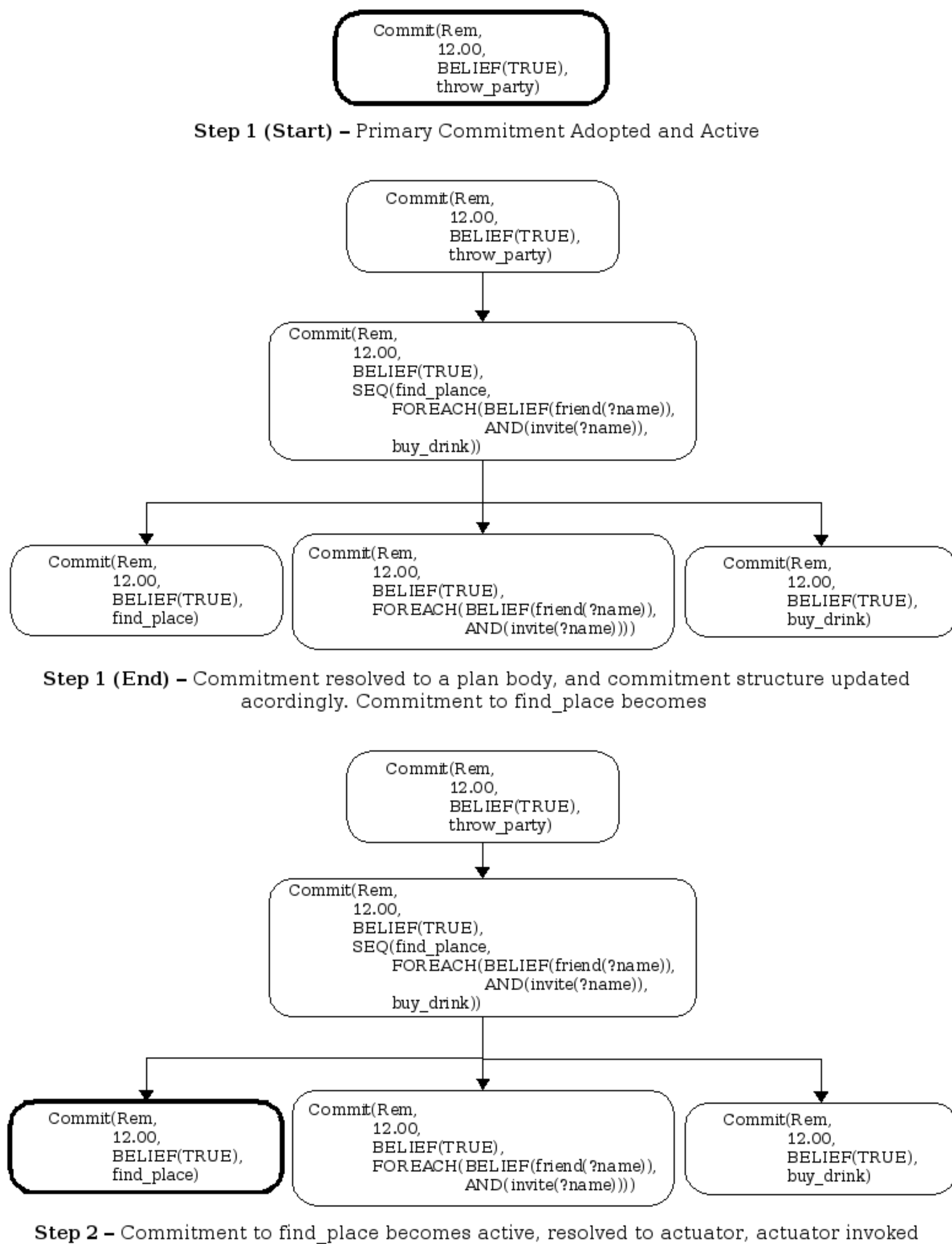
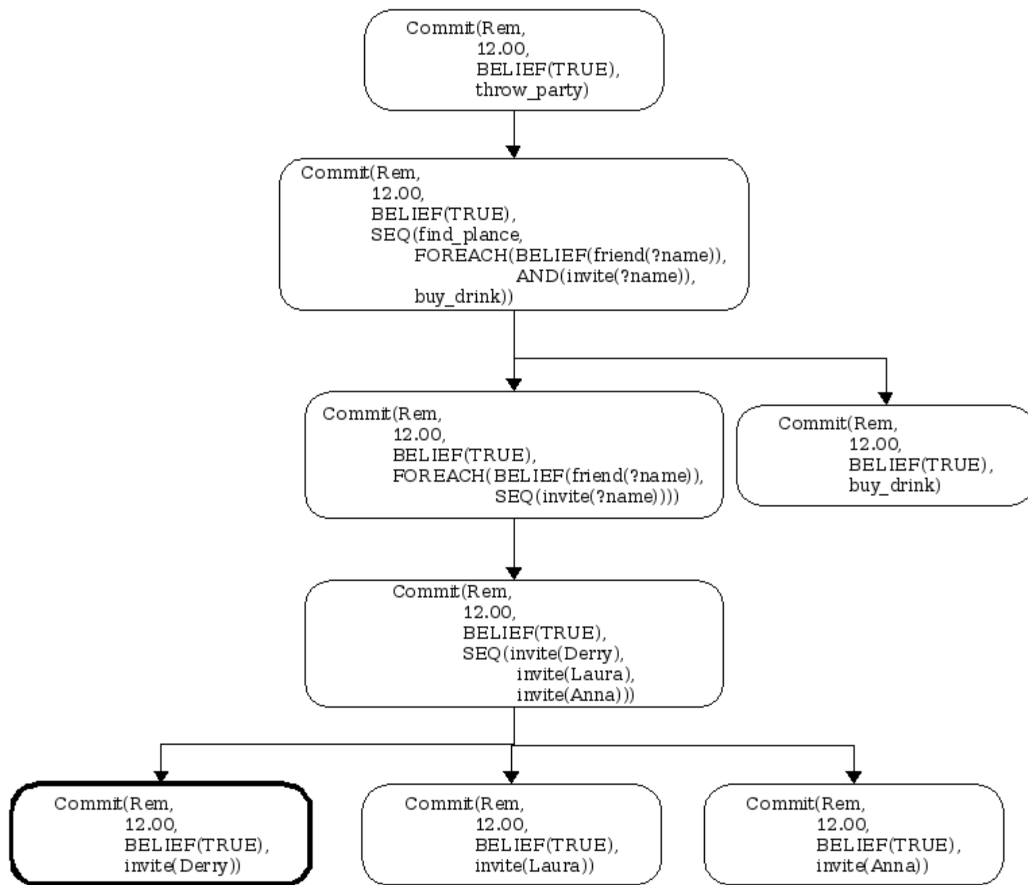


Figure 7.5: FOREACH Structure Expansion – Part 1



Step 3 – find_place returns true, hence fulfilling the commitment and causing it to be dropped. The commitment to the FOREACH construct then becomes active; is expanded to a secondary SEQ commitment, and the first child is set to true.

Figure 7.6: FOREACH Structure Expansion – Part 1

while the COMMITMENT is used to query the agent's commitment set. In addition to the four parameters present in the COMMIT construct, the COMMITMENT construct takes a fifth parameter – the state of the commitment.

```
COMMITMENT(?Self,?time),BELIEF(TRUE),action,state)
```

Commitment reasoning can be used in agent designs to guarantee that a commitment rule is not inadvertently fired twice. Thus, instead of AF-APL's old constructions:

```
BELIEF(X) & !BELIEFAboutAction => COMMIT(?Self,
                                         ?Now,
                                         BELIEF(TRUE),
                                         PAR(adoptBelief(AboutAction),
                                             doTheAction)
                                         );

SomeOtherConditions => COMMIT(?Self,
                              ?Now,
                              BELIEF(TRUE),
                              retractBelief(AboutAction)
                             );
```

a new cleaner construction can be formed using commitment reasoning:

```
BELIEF(X) & !COMMITMENT(?Self,?time,BELIEF(TRUE),DoTheAction,?state) =>
             COMMIT(?Self,
                    ?Now,
                    BELIEF(TRUE),
                    doTheAction);
```

7.4.1 Beliefs about Succeeded and Failed Commitments

Since the COMMITMENT construct includes the commitment's state, the COMMITMENT construct can be used to detect when commitments have failed or succeeded. For example, a commitment rule that is dependent on a commitment succeeding may take the form:

```
BELIEF(X) & !COMMITMENT(?Self,?time,BELIEF(TRUE),DoTheAction,SUCCEEDED) =>
             COMMIT(?Self,
                    ?Now,
                    BELIEF(TRUE),
                    doTheAction);
```

Chapter 8

Reactive Rules

The Reactive Rule allows an agent to invoke an action directly based on what it believes to be true. The reactive rule is simpler than a commitment rule, but is hence less robust. In this chapter we introduce the reactive rule, and compare it to its big brother, the commitment rule.

The Reactive Rule allows an agent to invoke an action directly based on what it believes to be true. The reactive rule is simpler than a commitment rule, but is hence less robust. A reactive rule can be used to code basic reflective behaviours into the agent's design. For example, a reactive rule to dodge an obstacle blocking a robot's progress could be encoded with:

```
BELIEF(blocked(ahead)) & BELIEF(moving(forward))
     => EXEC(dodgeObstacle);
```

Like a commitment rule, a reactive rule has two parts:

- **Belief Query Sentence** – A Query of the agents Belief Set. If the agent's beliefs entail this statement then the action, specified on the right hand side, will be invoked immediately. These queries cannot include details of the agent's goals or commitments; reactive rules are fired solely on the basis of beliefs, not higher level constructions.
- **Activity** – The actuator or plan which is to be attempted if the left hand side is entailed by the Agent's Belief Set.

There are a number of differences between the activities which can be committed to by a reactive rule, and those that can be committed to by their big brothers – the commitment rule. First, the reactive rule's activity is staticly bound at compile time; That is, a static plan is assigned to the reactive rule, guaranteeing that there is always a plan to be executed. Second, only the arrangement plan constructs (OR, XOR, AND, PAR, SEQ) may be used in reactive rules. The other constructs are expressly not allowed in reactive rules. Third, agents cannot reason about reactive rules, or the activities they are executing due to the firing of reactive rules. Despite these limitations, reactive rules

are powerful in their own way, since: first, that their static binding guarantees that the activity can be attempted immediately once the rule is triggered; second; that once triggered, reactive rules are managed solely in their own thread (similar to the firing of an actuator in its own thread), thus taking them out of the scope of the deliberation process.

In the following sections, some of the features of the reactive rule are discussed in greater detail.

8.1 Binding of Reactive Rules

Plans and actuators are more tightly bound to reactive rules then is the case for commitment rules. At the agent's birth time all actuator or plans referenced on the right hand side of a Reactive Rule must be bound to an actuator or plan declared in ALPHA code. The actions are then staticly linked to these actions; thus guaranteeing that whenever a Reactive Rule is fired, there will be appropriate actuators available.

During the Agent's Life time Action Definitions which are bound to Reactive Rules cannot be removed from the Agent's mental state. Reactive Rules can be removed from the Agent's mental state, thereby allowing previously bound Action definitions to be removed.

Reactive Rules can only be added to the agent's mental state if all actions referenced on the right hand side of the Reactive Rule are defined. That is, it may be necessary to add Actions to the Agent's mental state before a Reactive Rule can be added.

8.2 Firing of Reactive Rules

Reactive Rules are fired at the start of the agent execution cycle, straight after the perceptor firing phase, and before the commitment or goal reasoning phases; thus reactive rules are fired immediately after the agent's beliefs are updated.

Once triggered, the activity of a reactive rule is fired in its own thread – in an analogous way to the firing of actuators in commitment management. Thus, the stepwise treatment of commitment management as seen in the last chapter, is not applied to reactive rule execution.

8.3 Interference with Commitment Rules

A Reactive Rule is intended to give an agent a level of Reactive Control and as such may cause a behaviour which interferes with, or which is contradiction to a commitment previously held by an agent. Within the agent model actions which are the result of a Reactive Rule take precedence over actions which are being attempted as part of commitment management. The Commitment Manager must deal with interference from actions which have been triggered by Reactive Rules, but it is up to the agent's designer to try to ensure that reactive rules do not interfere with each other.

8.4 Reactive Rules & Reasoning

The reactive rule is inspired by reactive behaviours in humans, where an action is performed in direct response to an perceptual input. Thus, the reactive rules, and their execution is practically removed from the agent's deliberation process. This difference between the reactive rule and the other ALPHA constructs manifests itself in two ways: first the left hand side of a reactive rule cannot include any statements concerning commitments and goals; second, agents cannot reason about their reactive rules, or the reactive behaviours which have been invoked.

Chapter 9

Goals

A goal is a state of the world – or set of beliefs – that an agent wishes to bring about. In this chapter: we introduce the concept of a goal; show how goals are adopted by an ALPHA agent; and describe the means-end reasoning process, which attempts to determine a series of actions that might be used to achieve the goal.

A goal is a state of the world – or set of beliefs – that an agent wishes to bring about. Once a goal has been adopted by an agent, fulfilling that goal is a two stage process; Firstly, means-end reasoning to determine a plan that can achieve the desired goal; and secondly, the successful execution of this plan. We have already introduced commitment management as a process that can take a plan, and intelligently try to fulfil that plan. In this chapter, we look at how an agent can adopt explicit goals into its mental state, how an ALPHA agent attempts to derive plans for these goals.

The appropriate use of Goals can improve an agent's flexibility and intelligence. Through goals and means-end reasoning, an agent has some freedom to select an appropriate course of action to fulfil a goal. This flexibility is highly desirable in producing agents which can cope in complex dynamic environments.

Goals are related to the concept of a commitment, but are more broadly defined. As described in chapter 7 a commitment in ALPHA is a promise made by one agent to perform some action. Within ALPHA a Goal is a related concept, where the agent is trying to bring about a particular state of the world. Or more precisely the agent is trying bring about a particular set of world beliefs. Major differences between Goals and Commitments are that: (a) a goal is made with respect to a certain belief state rather than an action; (b) goals never become refined to other goals, instead they cause commitments to be adopted.

The agent can have many – and potentially conflicting – Goals that are contained in the Agent's Goal Set, as described in the next section. Goals can be added to the agent's commitment set through a number of plan operators, or through the declaration of Initial Goals. Once adopted, the agent will attempt to find plans or courses of action to achieve each of its goals. Even when a potential plan is found, commitments still need to be adopted and achieved, before the goal can be said to have been achieved. These are the processes of goal management, and are discussed in section 9.3. Finally,

in section 9.4 we show how an agent can directly reason on its own goals.

9.1 Goal Adoption

Although goals are an explicit construct of ALPHA, no initial goal constructs, or goal rule constructs are provided.

9.1.1 The ADOPT_GOAL Construct

The ADOPT_GOAL construct, described earlier, allows an agent to programmatically commit to adopting a GOAL. Once called, ADOPT_GOAL will immediately add the goal to the Goal Set, returning immediately.

```
BELIEF(requested(?someAgent,closeDoor)) =>
        COMMIT(?someAgent,
               Now,
               BELIEF(TRUE),
               ADOPT_GOAL(closed(door)));
```

Once adopted into the goal set, an agent will use means-end reasoning to attempt to determine a plan to achieve the goal. If such a plan can be determined, a secondary commitment to that plan structure will form as a child of the original goal. Fulfilling the plan then causes the original commitment to be dropped. The means-end reasoning process is, of course, performed asynchronously to guarantee the continued reactivity of the agent.

9.2 The Goal Set

At any time an ALPHA agent can have a number of different and potentially conflicting goals in its Goal Set. Unlike Beliefs, Goals naturally persist within the agent until they are: (a) explicitly dropped through within a plan; or (b) are realised successfully. The ways in which Goals may be added to, and removed from the Goal set are described in the following sections.

9.3 Goal Management

9.3.1 Goal Fulfilment

The fulfilment of goals is a three step process: firstly, means-end reasoning must be performed to obtain a plan which can potentially achieve the desired goal; secondly, this plan must be committed to; finally; if execution of the plan results in the goal being attained, then the goal is dropped by the agent.

For an ALPHA agent, we say that means-end reasoning, or planning, is the process whereby an agent attempts to find a sequence of actions, which, if executed, should bring the agent from its current mental state to the desired mental state, or goal. We will firstly present a description of the means-end reasoning process presently used in ALPHA, before moving on to show how this process fits into the agent's execution cycle.

The Planning Algorithm

The current planning algorithm is defined in commented pseudo-code in figure 9.2.

The algorithm is a recursive depth first search attempting to find a path from the goal state to the current state. The resultant series of actions, when executed in reverse order, will therefore bring the agent from the current state to the goal state.

Derived plans are only ever a sequencing of other plan bodies or actuators available to the agent. No disjunctions of plans are possible.

To see how the planning algorithm works, consider an agent with the following two plans available:

```
PLAN gotoDoor();
PRE BELIEF(TRUE);
POST BELIEF(am_at_door);
BODY SEQ(  locate(Door),
           moveTo(Door)
        );

PLAN leanAgainstDoor();
PRE BELIEF(am_at_door);
POST BELIEF(door_blocked);
BODY SEQ(  positiionAgainst(Door),
           stop
        );
```

If the agent were to adopt a goal to BELIEF(door_blocked), i.e. the agent held the goal:

```
    GOAL(Self,Now,BELIEF(TRUE), BELIEF(door_blocked))
```

then the means-end reasoner would derive the simple plan body `SEQ(gotoDoor,leanAgainstDoor)`.

Commitment Adoption

Once a plan has been developed by the means-end reasoner, the goal management process builds a commitment to achieve the derived plan and adds it to the agent's set of primary commitments.

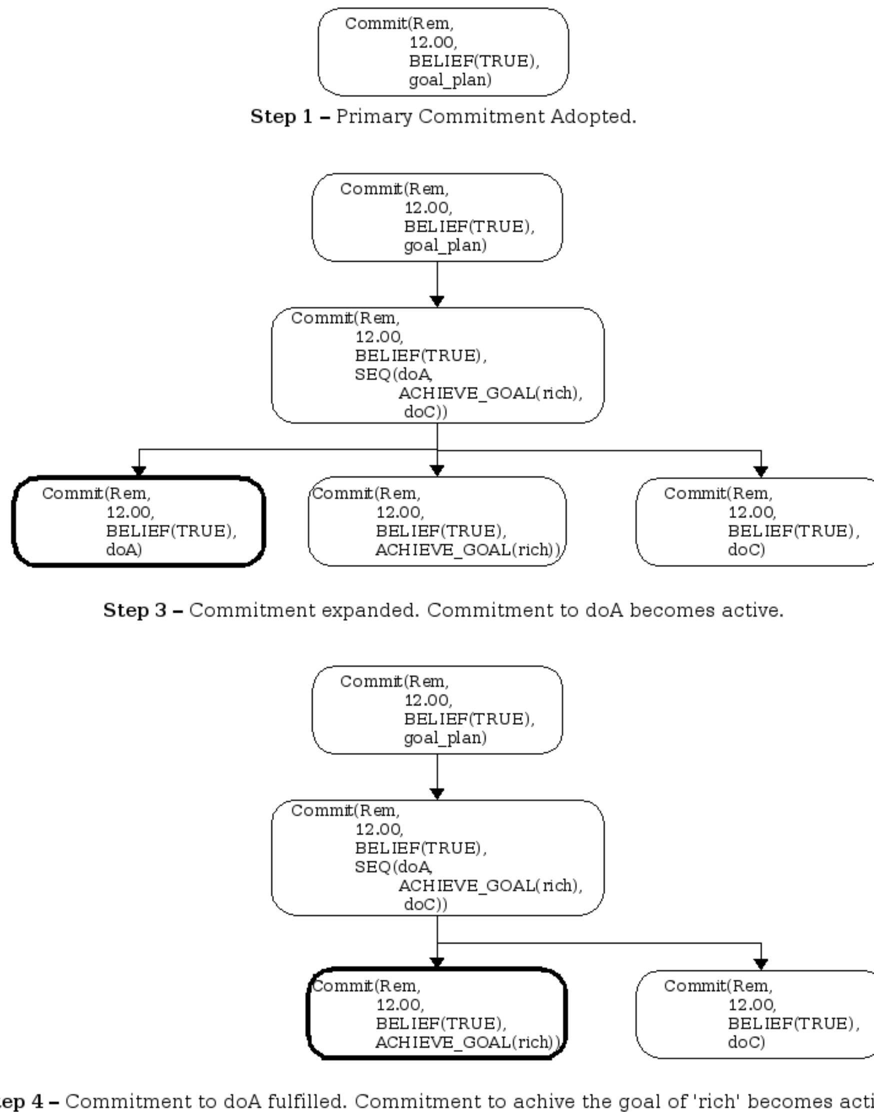


Figure 9.1:

```
Vector constructPlan(inGoal) {
   // a vector of FOS elements -- This will be the list of actions returned.
  Vector returnVector = new Vector();
  // Establish list of plans and actuators
  List availableActivities = all_static_plans + all_actuators;
  // Iterate through each of the agents plans
  while(have_more_activites && no_result_found){
     Activity nextActivity = availableActivites.getNext();
     BeliefSentence post_condition = nextPlan.getPostCondition();
     // if the post condition of the activity matches the current goal
     if(post_condition == inGoal) {
        // The currently examinded activity achieves the required goal, but
        // can the activity be called from the current state.
       if(nextActivity.getPreConditions == Current_state) {
           // We have identifed a plan which will take us from the current
           // state to the goal state. Add this to the plan structure being
           // returned. Break out of the while loop.
           no_result_found = false;
       }
       // else we still have not reached the final state,
       else {
           // recursivly call this method to try to find a plan structure
           // which can bring us from the current state to the pre-condition
           // of the activity currently being examined.
           result = constructPlan(nextActivity.getPreCondition());
           // If no result was found by the recuursive call, then give up on
           // this branch
           if(result == null) {
              // the currently examined activity is a dead end. Move on.
           }
           // else we got a result
           else {
             // the recusrive call was able to generate a plan body which
             // brings us from the current state to the pre-condition of the
             // currently examined plan. We have successfully found a plan
             result.add(nextActivity.getIdentifier());
             returnVector = result;
             no_results_found = false;
          }
       }
     }
     // else if the post condition of the activity does not match the goal
     else {
          // then move onto the next activity
     }
   }
   return returnVector;
}
```

Figure 9.2: The basic ALPHA planning algorithm

9.3.2 Goal Retraction

Once adopted, goals persist in the agent's mental state until one of the following three possibilities occur:

- **Fulfilment of Goal** – If at any time after adopting the Goal, the agent believes that the Goal has been met – the agent's Belief Set entail the Goal – then the Goal will be dropped along with any remaining resultant commitments.
- **Explicit Retraction** – An in-built *retractGoal* actuator is provided to allow for the programmatic retraction of Goals. Any resultant commitments will also be dropped.

9.3.3 Goal Management within the Agent Execution Cycle

The goal management process begins following the reactive rule management process. The agent first analyses its current set of goals to determine if any have been met; if the agent's belief set entails any of its goals, then those goals are dropped by the agent. Any commitments adopted by the agent to achieve one of these goals are immediately dropped. Secondly, the agent begins the means-end analysis towards each of its goals. The planning process is spawned off in its own thread.

Discussion of the Goal Process

A drawback to the use of an asynchronous planning model is that by the time the planning process has returned, the plan may be invalidated by changes in the real world or agent state. This is a reality of the real world, and just something that commitment management and re-planning will need to attempt to overcome.

Also, the algorithm is computationally expensive. But since the algorithm is run in parallel to the more reflective and reactive abilities of the agent, the impact on an agent's reactivity is negligible.

9.4 Reasoning about Goals

The agent can explicitly reason about its own goals in Belief Rules and Commitment Rules. For example, a belief rule that is dependent on

```
BELIEF(X) & GOAL(close(door)) =>
            COMMIT(Self,Now,BELIEF(TRUE),doSomeAction());
```

Chapter 10

Modules

Modules provide agents with data and functionality store that are not appropriate for representation or implementation at the knowedge based intentional level. This chapter introduces the MODULE construct, showing how it can be used by perceptors and actuators to share data and functionality.

In ALPHA a module provides a simple way of declaring an external block of code (Java class or C object/library) which can be used to share functionality and memory between different actuators and perceptors. The construct takes one argument, the name of the module to be loaded by an agent. For example, a Java library which implements a hashmap, might be declared with the following:

```
MODULE ie.ucd.myProject.modules.MyHashMap.class;
```

In the case of a Java module, the class is constructed once at start time, and is then available for use by actuators and perceptors as needed. For C code, an init() function must be present to initialise the module for later use.

Beyond the declaration of modules, there is no special significance to modules in ALPHA. They are only intended as an aid to external actuator and perceptor designing. Hence, they are not reasoned about by the agent, they have no intelligence, nor do they have any effect on any of the other ALPHA constructs.

Chapter 11

Some Other Language Features

We will now look at some of the more traditional features of the languages, including comments and namespace.

11.1 The ALPHA Namespace

To aid code re-use and rapid prototyping, ALPHA uses a namespace convention similar to that used by Java.

Some differences exist between the Java namespace and the ALPHA namespace. These differences are due to ALPHA's support for a number of different types of code and files. An actuator and plan can exist in the same directory and with similar names. For example it is perfectly acceptable to have both:

```
ie.ucd.myProject.ask.plan
```

and

```
ie.ucd.myProject.ask.class
```

where these are represented in ALPHA with the similar identifiers:

```
ACTUATOR ie.ucd.myProject.ask;
PLAN ie.ucd.myProject.ask;
```

11.2 Calling the Correct Actuator

In commitments or plans there is no need to identify actuators or plans by their full identifier – that is, as long as there are no ambiguous identifiers. For example, if an agent has an actuator with the following identifier:

```
IDENTIFIER ie.ucd.myProject.teaMaking2.boilKettle;
```

then that actuator can be identified simply as boilKettle in any plan body. However, if the agent also had another actuator defined with the same name, but different package:

```
IDENTIFIER ie.ucd.myProject.teaMaking2.boilKettle;
```

then it would be necessary to explicitly define which boilKettle actuator was required. Similarly, if the agent only had the first actuator, but also had a plan with the same or similar identifier:

```
IDENTIFIER ie.ucd.myProject.teaMakingPlans.boilKettle;
```

then it would be necessary to explicitly identify either the actuator or plan fully.

11.3 Comments

ALPHA supports single line comments declared with by \\. The ALPHA compiler will ignore these characters and the rest of the line that follows. There is only one exception to these comment rules. If the comment identifiers appear within a string constant i.e. between a set of double quotes then they will be treated as parts of the string constant and will not be ignored by the comment parsing process. This is similar to the behaviour in Java.

Part II

Using ALPHA

Chapter 12

Roles

Roles allow us to group sets of agent constructs together to form either prototype agent designs, or to allow an agent to activate or deactivate certain abilities at runtime. In this chapter: we describe the ROLE macro; show how it can be used to give a form of ineritence in rapid agent prototyping; and describe how it can also be used by the agent at runtime.

Roles can be viewed in either a top-down or bottom up way: from a bottom-up view, roles allow us to bundle up the capabilities of an agent into manageable groups, which can be reasoned about; from a top-down perspective, roles allow agents to be rapidly prototyped from role designs.

Roles are not strictly part of ALPHA. Instead, they are macros that are replaced with pure ALPHA code at run-time.

12.1 Rapid Prototyping with Roles

A role is a high level agent design that can be compiled into an agent type by the ALPHA compiler. These agent types can then be instantiated to a number of different agents by the ALPHA interpreter.

In a role file, the developer can specify the actuators, perceptors, and mental state of an agent. The designer can also call for the inheretance of other role files with the INCLUDE_ROLE macro as follow:

```
INCLUDE_ROLE ie.ucd.core.fipa.role.FIPARole;
```

When compiling the resultant ALPHA agent description, the compiler will search for the following role description in the project's classpath:

```
ie.ucd.agents.agent1.rle
```

Using the INCLUDE_ROLE macro in this way, is similar to the use of a #include in C or C++.

Chapter 13

Compiling ALPHA Code for Interpretation

In this chapter we introduce roles, plan files, and a number of other auxiliary *files, which can be used to compile a complete ALPHA agent for interpretation.*

ALPHA as described informally in Part I can be written directly and interpreted by the ALPHA compiler, but it is often convenient to make use of the ALPHA compiler to perform some mundane syntax and semantics checking.

13.1 The ALPHA Compilation Process

The ALPHA Compiler takes agent descriptions built from role files (.rle), actuators and plan descriptors, and builds them into complete agent designs. Figure 13.1 illustrates this process. Once compiled, ALPHA agent designs (.alpha) can be instantiated to any number of individual agents by the ALPHA interpreter. An agent designer will typically work with the higher level role files (.rle) rather than lower level ALPHA code.

13.2 The ALPHA Auxiliary Files

13.2.1 Role Files

Role files are a superset of ALPHA, adding a number of macro rules that are compiled out of the code by the ALPHA compiler. Specifically, ALPHA roles add `USE_ROLE`, `ACTUATOR`, `PERCEPTOR`, and `PLAN` to the list of allowed statements.

- **USE_ROLE** This construct takes one argument which references some other role file (.rle) available for behaviour by the Compiler. The behaviour of this construct is similar to the #include pre-compiler command in C.

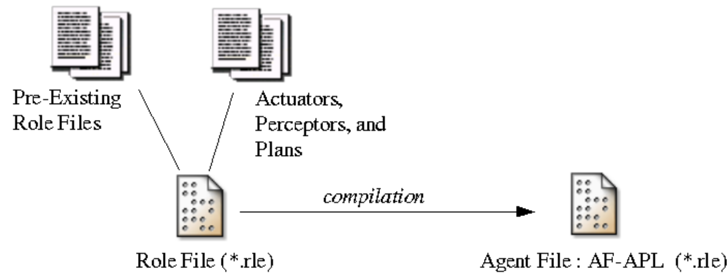


Figure 13.1: Compiling Roles to ALPHA

- **ACTUATOR** The ACTUATOR construct takes one argument, i.e., an actuator identifier. This actuator identifier should match an actuator description file (.act) that is sourcable by the ALPHA compiler. The arity of the actuator identifier included in the role file must match the arity of the identifier in the actual actuator description file, but the name of the variables need not match. This construct is replaced with the contents of the actuator descriptor file in the compiled ALPHA agent type.
- **PERCEPTOR** This constructs explicitly states the name of an actuator to be included in the compiled agent design. Specifically, the identifier is assumed to be the name of a Java class that implements the perceptor. The compiler will add a full perceptor declaration to the resultant ALPHA agent type.
- **PLAN** This construct takes one argument which is a complete plan identifier. This plan identifier should match a plan description file (.plan) which is sourcable by the ALPHA compiler. The arity of the plan identifier included in the role file must match the arity of the identifier in the actual plan description file, but the name of the variable need not match. This construct will be replaced with the complete plan description in the compiled ALPHA agent type.
- **LOAD_MODULE**
- **Mental State Constructs** These are the standard ALPHA mental constructs including Beliefs, Belief Inference Rules and Commitment Rules.

13.2.2 Plan Description Files

Plan Files currently have the following general format:

```
BEGIN{PLAN}
IDENTIFIER PLAN_IDENTIFIER;
PRE BELIEF_SENTENCE;
POST BELIEF_SENTENCE;
BODY PLAN_BODY;
END{PLAN}
```

All actuators explicitly used within the plan must be explicitly included at the top of the plan file. The remainder of the plan file format matches the ALPHA plan construct described in section 6.1.

13.2.3 Actuator Description Files

Actuator description files define actuators which are to be included in the initial agent design. The general format of an actuator description is shown below and is identical to the Actuator Construct in ALPHA (Section 5).

```
BEGIN{ACTION}
IDENTIFIER ACTUATOR_IDENTIFIER;
PRE BELIEF_SENTENCE;
POST BELIEF_SENTENCE;
ACTUATOR CODE_IDENTIFIER;
END{ACTION}
```

13.3 ALPHA Roles & AF-APL

As described earlier, ALPHA is a direct derivitive of the AF-APL language and some effort has been made to ensure backward compatibility with AF-APL. Specifically, AF-APL role files (.rle) are valid ALPHA role files. The resultant agent design will be fully compliant ALPHA code, but there may be some differences between the behaviour of an ALPHA agent based on a particular role file, and the behaviour of an AF-APL agent based on the same role file. In particular, the commitment management process used by the two languages is subtilty different. As long as specific synchronisation of behaviour is not required, it is likely that ALPHA and AF-APL agents will produce the same result.

Chapter 14

Example Code

For illustration, we now present some code for a typical scenario that has influenced AF-APL's current features. Figure 14.1 presents a program fragment for an office assistant robotic agent. Starting from the top, we first see the use of the USE_ROLE macro to import AF-APL roles that provide domain specific definitions for FIPA compliant communication, and control of an autonomous wheelchair robot. The USE_ACTUATOR macro is then used to import a pre-defined actuator for vocalising utterances. This is followed by an in-line plan definition to deliver a parcel to a destination. The plan makes use of the TRY_RECOVER plan operator to initiate social help for recovery in the event that the basic delivery plan fails. The basic delivery plan also makes use of the DO_BEHAVIOUR_UNTIL operator which will cause the agent to follow the wall until it believes that a door has been found.

A commitment rule used states that if the agent believes that it has been asked to deliver a package, and if it believes that a superior made the request, and that the agent is not already committed to some other task, then the agent should immediately blindly commit to a simple plan. The plan uses an actuator inherited from the FIPARole to send an acknowledgement message to the requesting agent, before using deliver plan, to perform the delivery task. Upon arrived at the destination, the agent will announce its presence. In addition to the commitment rule, a reactive rule is also specified; this simple rule states that if the agent believes there is an object blocking its active path, then it will initiate a reactive action to dodge the obstacle.

```
1  USE_ROLE ie.ucd.core.fipa.role.FIPARole;
2  USE_ROLE de.tzi.RollandRole;
3  USE_ACTUATOR de.tzi.MARY.speak(?utterance);
4
5  // Explicit Plan Definition
6  BEGINPLAN
7  IDENTIFIER de.tzi.safeDeliver();
8  PRE BELIEF(TRUE);
9  POST BELIEF(delivered(parcel));
10 BODY TRY_RECOVER(SEQ(DO_BEHAVIOUR_UNTIL(followWall,
11                                         BELIEF(found(Door))),
12                     enterDoor),
13                 social_recovery),
14 ENDPLAN
15
16 // Commitment Rule
17 BELIEF(requested(?agent,deliver(?object,?destination)))
18 & BELIEF(isSuperior(?agent)) & !COMMIT(?a,?b,?c,?d)
19               => COMMIT(?agent,
20                         ?Now,
21                         BELIEF(TRUE),
22                         SEQ(fipaSend(?agent,inform(ack(deliver(?object,?dest)))),
23                             safeDeliver,
24                             speak("I have a parcel for collection")
25                             )
26                         );
27
28 // Reactive Rule
29 BELIEF(blocked(ahead)) & BELIEF(moving(forward)) => EXEC(dodgeObstacle);
```

Figure 14.1: Sample AF-APL Program

Bibliography

[1] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa. JADE — A FIPA-compliant agent framework. In *Proceedings of the 4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99)*, pages 97–108, London, UK, 1999. The Practical Application Company Ltd.

[2] M. Bratman, D.J. Israel, and M.E. Pollock. Plan and resource-bounded practical reasoning, 1998. Computational Intelligence 4(4), pp349-355,.

[3] M.E. Bratman. *Intentions, Plans, and Practical Reason*. Harvard University Press, Cambridge, MA, USA, 1987.

[4] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents Components for Intelligent Agents in Java. *AgentLink Newsletter*, 1, 1999.

[5] Rem W. Collier. *Agent Factory: A Framework for the Engineering of Agent Oriented Applications*. PhD thesis, University College Dublin, 2001.

[6] Rem W. Collier, G.M.P. O'Hare, Terry Lowen, and C.F.B. Rooney. Beyond prototyping in the factory of the agents. In *3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03)*, Prague, Czech Republic, 2003.

[7] M. Dastani, F. Dignum, and J.J. Meyer. 3APL: A Programming Language for Cognitive Agents. ERCIM News, European Research, 2000. Consortium for Informatics and Mathematics, Special issue on Cognitive Systems, No. 53,.

[8] Mehdi Dastani, Birna van Riemsdijk, Frank Dignum, and John-Jules Meyer. A Programming Language for Cognitive Agents: Goal Directed 3APL. In *Proceedings of AAMAS 03*, 2003.

[9] Daniel C. Dennett. *The Intentional Stance*. The MIT Press, Massachusetts, 1987.

[10] Daniel C. Dennett. *Kinds of Minds: Toward an Understanding of Consciousness*. Basic Books, New York, 1996.

[11] M.P. Georgeff and F.F. Ingrand. Decision-making in an embedded reasoning system. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 202–206, Detroit, Michigan, USA, 1989.

[12] M.P. Georgeff and A.L. Lansky. Reactive reasoning & planning. In *Proceedings of the Sixth Intenational Conference on Artificial Intelligence (AAAI-87)*, pages 677–682, Seatle, WA, USA, 1987.

[13] Afsaneh Haddadi. *Communciation and Cooperation in Agent Systems: A Pragmatic Theory*. Number 1056 in Lecture Notes in Computer Science. Springer-Verlag: Heidelberg, Germany, 1996.

[14] Koen V. Hindrikis, Frank de Boer, Wibe van der Hoek, and John Jules Meyer. Failure, Monitoring and Recovery in the Agent Langauge 3APL. In *Proceedings of AAAI 1998 Fall Symposium on Cognitive Robotics*, 1998.

[15] D. Martin, A. Cheyer, and D. Moran. The Open Agent Architecture: a framework for building distributed software systems. *Applied Artificial Intelligence*, 13(1/2):91–128, 1999.

[16] H. Nwana, D. Ndumu, L. Lee, and J. Collis. ZEUS: A Tool-Kit for Building Distributed Multi-Agent Systems. *Applied Artificial Intelligence Journal*, pages 129–186, 1999.

[17] A. Rao. AgentSpeak(L): BDI Agents speak out in a logical computable language. In *proceeedings of the Seventh European Workshop on Modelling autonomous agents in a MultiAgent world*, Institute for Perception Research, Eindhoven, The Netherlands, 1996.

[18] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, April 1991.

[19] Anand S. Rao and Michael P. Georgeff. BDI agents: from theory to practice. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319, San Francisco, CA, USA, 1995. The MIT Press: Cambridge, MA, USA.

[20] Yoav Shoham. Agent oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[21] S.R. Thomas. The PLACA agent programming language. In *The Proceedings of Agent Theories, Architectures, and Languages*, pages 356–370, Amsterdam, 1994.

[22] M. Wooldridge. *Reasoning about Rational Agents*. Intelligent Robots and Autonomous Agents. The MIT Press, Cambridge, Massachusetts, 2000.

[23] Michael Wooldridge and Paolo Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In *AOSE*, pages 1–28, 2000.